CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS
DEL INSTITUTO POLITÉCNICO NACIONAL

UNIDAD ZACATENCO
DEPARTAMENTO DE COMPUTACIÓN

"Development of Artificial Intelligence Techniques for Playing Chess Computer"

A dissertation submitted by
Eduardo Vázquez Fernández

For the degree of
Doctor in Computer Science

Advisors
Dr. Carlos Artemio Coello Coello
Dr. Feliú Davino Sagols Troncoso

Mexico City, Mexico December, 2012

CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS
DEL INSTITUTO POLITÉCNICO NACIONAL

UNIDAD ZACATENCO
DEPARTAMENTO DE COMPUTACIÓN

"Desarrollo de Técnicas de Inteligencia Artificial para Jugar Ajedrez por Computadora"

Tesis que presenta
Eduardo Vázquez Fernández

Para obtener el grado de
Doctor en Ciencias en Computación

Directores de tesis
Dr. Carlos Artemio Coello Coello
Dr. Feliú Davino Sagols Troncoso

México, Distrito Federal — Diciembre, 2012

Eduardo Vázquez Fernández: *"Development of Artificial Intelligence Techniques for Playing Chess Computer"*, December, 2012.

A mi núcleo familiar más importante **Grisel**, **Gaby** y **Dany**

AGRADECIMIENTOS

Deseo expresar mi agradecimiento a diferentes instituciones y personas que hicieron posible la realización del presente trabajo.

Deseo agradecer a mis asesores, el Dr. Carlos A. Coello Coello y el Dr. Feliú D. Sagols Trocoso, su confianza, orientación, conocimientos, y sobre todo, su dirección del presente trabajo doctoral.

Deseo agradecer al Dr. Gerardo De la Fraga, a la Dra. Nareli Cruz , al Dr. Andrés Gómez y al Dr. Ricardo Landa por leer esta tesis y por sus valiosos comentarios al respecto.

Deseo agradecer a mi núcleo familiar más importante: mi esposa Grisel y mis hijos Gaby y Dany por ser el motor que impulsó la realización del presente trabajo.

Quiero agradecer a mis padres sus enseñanzas y cuidados durante diversas etapas de mi vida.

A mis seres queridos, de manera muy especial a la memoria de mi hermano Fidel. A tia Loren y mis hermanos Lissette, Alejandra, Isaac y Miguel.

Quiero agradecer a mis compañeros de doctorado las vivencias amenas y los conocimientos académicos compartidos. Muy especialmente a: Alfredo Arias, Cuauhtémoc Mancillas, Arturo Yee, Lil Mariaf Rodríguez, Sandra Díaz, Adriana Lara, Adriana Menchaca, Antonio López, Saúl Zapotecas, Julio Barrera, Lourdes López, por citar sólo algunos.

Agradezco al personal secretarial del departamento de Computación, Sofía Reza, Felipa Rosas y Erika Ríos por su valioso e incondicional apoyo en diversos trámites administrativos.

Deseo agradecer al Instituto Politécnico Nacional y a la Escuela Superior de Ingeniería Mecánica y Eléctrica, Unidad Culhuacán, por brindarme las facilidades y apoyos necesarios para el desarrollo del presente trabajo.

Agradezco al CONACyT la beca otorgada durante la realización de estos estudios de doctorado, y muy especialmente al CINVESTAV por ofrecerme un ambiénte académico de calidad y excelencia.

Este trabajo de tesis se derivó del proyecto CONACyT titulado "Escalabilidad y Nuevos Esquemas Híbridos en Optimización Evolutiva Multiobjetivo" (Ref. 103570), cuyo responsable es el Dr. Carlos A. Coello Coello.

Agradezco sinceramente a todas aquellas personas que de alguna manera contribuyeron y apoyaron el desarrollo del presente trabajo, pero que por motivo de espacio sus nombres han sido omitidos.

ABSTRACT

Basically, a chess engine is composed of a move generator, a search algorithm of the principal variation and an evaluation function. In this work, we designed and implemented a chess engine by adding knowledge to the evaluation function through artificial intelligence techniques.

The research had three main contributions. In the first, we proposed a neural network architecture to obtain the positional values of chess pieces in a way analogous to human chess players. With this proposal, our chess engine reached a rating of 2178 points. In the second contribution, we report an evolutionary algorithm which has a selection mechanism that favors virtual players that are able to "visualize" (or match) more moves from those registered in a database of chess grandmaster games. This information is used to adjust the basic weights set of the evaluation function of the chess engine. This proposal does not attempt to increase level of play in our chess engine, but instead aims to deduce the known values from chess theory for this basic weights set.

Finally, in the third contribution of this work, we used two steps to carry out the weights adjustment of our chess engine. In the first step, we performed an exploration search through the previous evolutionary algorithm, but now we adjust a larger number of weights (from five to twenty eight). With this change, we obtained an increase in the rating of the chess engine from 1463 to 2205 points. In the second step, we used the Hooke-Jeeves algorithm to continue the weights adjustment for the best virtual player obtained in the previous step. Using this algorithm as a local search engine, we increased the rating of our chess engine from 2205 to 2425 points.

RESUMEN

Básicamente, un motor de ajedrez está compuesto de tres partes: un generador de movimientos, un algoritmo de búsqueda de la variante principal y una función de evaluación. En este trabajo diseñamos e implementamos un motor de ajedrez agregando conocimiento a la función de evaluación a través de técnicas de inteligencia artificial como lo son los algoritmos evolutivos y/o las redes neuronales.

La investigación versa sobre tres contribuciones principales. En la primera, proponemos una arquitectura de redes neuronales para obtener los valores posicionales de las piezas de ajedrez. Con esta propuesta, nuestro motor de ajedrez alcanzó una calificación de 2178 puntos en el sistema de medida empleado por la Federación Internacional de Ajedrez. En la segunda contribución, proponemos un algoritmo evolutivo con un mecanismo de selección basado en juegos de grandes maestros para ajustar el conjunto básico de pesos del motor de ajedrez.

Finalmente, en la tercera contribución, usamos dos pasos para llevar a cabo el ajuste de pesos. En el primero, realizamos la búsqueda de exploración a través del algoritmo evolutivo previo, pero tomando en cuenta un mayor número de pesos (de cinco a veintiocho). Con este cambio obtuvimos un incremento en la calificación de nuestro motor de ajedrez de 1463 a 2205 puntos. En el segundo paso, usamos el método de Hooke-Jeeves para continuar ajustando los pesos del mejor jugador virtual obtenido en el paso anterior. Usando este algoritmo como un buscador local, logramos incrementar la calificación de 2205 a 2425 puntos.

CONTENTS

1 INTRODUCTION 1
- 1.1 Statement of the problem 1
- 1.2 Hypothesis 1
- 1.3 Objectives 1
- 1.4 Contents of the document 2

2 COMPUTER CHESS 5
- 2.1 Computer chess history 5
- 2.2 Notions and concepts 14
  - 2.2.1 Game tree 15
  - 2.2.2 Search tree 16
- 2.3 Fundamental components 16
- 2.4 Board representation and move generation 17
- 2.5 Search algorithms 18
  - 2.5.1 Minimax 18
  - 2.5.2 Negamax 19
  - 2.5.3 Branch-and-bound algorithm 20
  - 2.5.4 Alpha-beta pruning 22
  - 2.5.5 Quiescence search 22
  - 2.5.6 Iterative deepening 23
- 2.6 Evaluation function 24
- 2.7 Our chess engine 26
- 2.8 Final remarks of this chapter 27

3 SOFT COMPUTING IN CHESS 29
- 3.1 Artificial neural networks 29

3.1.1 A short history of neural networks 29
3.1.2 Basic concepts of artificial neural networks 31
3.1.3 Activation function types 33
3.1.4 Advantages and disadvantages of neural networks 34
3.1.5 Neural networks architecture 35
3.1.6 Learning process 36
3.2 Evolutionary algorithms 38
3.2.1 A short review of evolutionary algorithms 38
3.2.2 Components of an evolutionary algorithm 39
3.2.3 Evolutionary algorithms versus mathematical programming techniques 42
3.2.4 Evolutionary computation paradigms 43
3.3 Differential Evolution 47
3.3.1 Initialization of vectors 47
3.3.2 Mutation 48
3.3.3 Crossover 49
3.3.4 Selection 49
3.3.5 DE Family of Storn and Price 49
3.4 Previous Related Work 50
3.4.1 Works related to unsupervised adjustment 51
3.4.2 Works related to supervised adjustment 56
3.4.3 Works related to hybrid adjustment 57
3.5 Final Remarks of this chapter 58
4 TUNING WEIGHTS THROUGH A NEURAL NETWORK ARCHITECTURE 63
4.1 Introduction 63
4.2 Evaluation function 63
4.2.1 Material values of the chess pieces 64
4.2.2 Positional values of the chess pieces 64
4.3 Methodology 65
4.3.1 Neural network architecture 65
4.3.2 Components of our evolutionary algorithm 74
4.3.3 Our evolutionary algorithm 76
4.4 Experimental design 77
4.5 Experimental results 77
4.5.1 Experiment A 77
4.5.2 Experiment B 78
4.5.3 Discussion of the results 81
4.6 Final remarks of this chapter 82

5 TUNING WEIGHTS WITH A DATABASE OF CHESS GRANDMASTER GAMES 85
5.1 Introduction 85
5.2 Chess engine 86
5.3 Methodology 86
5.3.1 Components of our evolutionary algorithm 87
5.3.2 Evolutionary algorithm 88
5.3.3 Database of games 89
5.4 Experimental results 90
5.4.1 Tuning weights 90
5.4.2 Additional Games 93
5.5 Final remarks of this chapter 96
6 TUNING WEIGHTS WITH THE HOOKE-JEEVES METHOD 97
6.1 Evaluation function 97
6.1.1 King's positional value 98
6.1.2 Queen's positional value 99
6.1.3 Rook's positional value 99
6.1.4 Bishop's positional value 100
6.1.5 Knight's positional value 101
6.1.6 Pawn's positional value 102
6.2 Methodology 102
6.2.1 Components of our evolutionary algorithm 102
6.2.2 Phases of our method 103
6.2.3 Initialization 106
6.2.4 Database of games 106
6.3 Experimental results 106
6.3.1 First experiment 106
6.3.2 Second experiment 109
6.3.3 Third experiment 111
6.4 Final remarks of this chapter 112
7 CONCLUSIONS AND FUTURE WORK 115
A ELO RATING SYSTEM 117
A.1 Elo formula 117
A.2 World chess federation 120
B UCI PROTOCOL 123
B.1 From GUI to chess engine 124
B.2 From chess engine to GUI 125

LIST OF FIGURES

Figure 1 Alan Turing 6
Figure 2 Claude Shannon 7
Figure 3 The Univac Computer *Maniac I*. 8
Figure 4 The chess computer *Belle*. 11
Figure 5 *Deep Thought's* team. 12
Figure 6 Kasparov vs *Deep Blue*. 13
Figure 7 *Hydra* super-computer. 14
Figure 8 Nodes description in the game tree. 15
Figure 9 Board representation with the 0×88 method. 17
Figure 10 Example of operation of the minimax algorithm. 19
Figure 11 Example of operation of the negamax algorithm. 19
Figure 12 If $F(P_1) = -1$, then $F(P) \geqslant 1$ and we do not have to know the exact value of $F(P_2)$ if we can deduce that $F(P_2) \geqslant -1$. This happens if $F(P_{21}) \leqslant 1$. 21
Figure 13 Example of pruning with the branch-and-bound algorithm. 21
Figure 14 Example of pruning with the alpha-beta algorithm. 24
Figure 15 Example diagram. 25
Figure 16 Architecture of our chess engine. 28
Figure 17 Model of a neuron. 32
Figure 18 Feedforward network with a single layer of neurons. 59
Figure 19 Network with one hidden layer and one output layer. This network is fully connected. 60

Figure 20 Recurrent network. 61
Figure 21 Charles Darwin. 61
Figure 22 Main stages of the differential evolution algorithm. 62
Figure 23 Position to illustrate feature extraction. 65
Figure 24 Neural networks architecture used in the evaluation of the pieces' positional values. 67
Figure 25 Position to illustrate feature extraction. 68
Figure 26 The white queen prevents checkmate on f1 square. 69
Figure 27 The value of the queen of g3 is greater than the value of the queen on a8. 70
Figure 28 The white rook on the seventh row permits to the white side win the black queen. 70
Figure 29 The black king receives checkmate by the white rooks on b7 and c7. 71
Figure 30 The value of the bishop on e5 is greater than the value of the bishop on c8. 71
Figure 31 The value of the knight on d6 is greater than the value of the knight on g6, and this is greater than the value of the knight on b8. 75
Figure 32 Position to illustrate feature extraction. 76
Figure 33 Flowchart of the evolutionary algorithm adopted in this work. 78
Figure 34 Histogram of wins, draws and losses for the best virtual player at generation 0 ($player0$) against Rybka 2.3.2a. 81
Figure 35 Histogram of wins, draws and losses for the best virtual player at generation 50 ($player_{50}$) against Rybka 2.3.2a. 82
Figure 36 Chromosome adopted in our evolutionary algorithm. 87
Figure 37 Flowchart of our proposed evolutionary algorithm. 90
Figure 38 Average weight values of the population during 50 generations. 92
Figure 39 Standard deviation of the weights in the population during 50 generations. 93
Figure 40 Final position for the game between the human player ranked at 1600 points (with white pieces) versus "average weights in generation 50" (with black pieces). 96

Figure 41 Evolutionary process for the exploration search. The plot shows the number of positions solved (a total of 1000) for the best virtual player and the average weight values of the 20 virtual players during 200 generations. 109
Figure 42 Histogram of wins, draws and losses for $Chessmaster_{2500}$ against $VP_{exploitation}$ (H1), $VP^{200}_{exploration}$ (H2), $VP^{0}_{exploration}$ (H3). 112
Figure 43 Rating difference versus percentage score. This figure was taken from http://www.chessbase.com/newsdetail.asp?newsid=7114. 120
Figure 44 Comparison of the Elo's prediction and better prediction. This figure was taken from http://www.chessbase.com/newsdetail.asp?newsid=562. 121

LIST OF TABLES

Table 1 History of the ACM North American Computer Chess Championship. 9
Table 2 History of the World Computer Chess Championship. 10
Table 3 Results of the match Kasparov vs Deep Blue (1996). 12
Table 4 Results of the match Kasparov vs Deep Blue (1997). 12
Table 5 Main features of the three main evolutionary computation paradigms. 48
Table 6 Symbols for chess position assessment. 56

Table 7 Initial weight values of black pawns than obstruct the black bishop's movement. 73
Table 8 Initial weight values of the white pawns than obstruct the black bishop's movement. 74
Table 9 Number of games won, drawn and lost for the best virtual player at generation 50 against the best virtual player at generation 0. 79
Table 10 Ratings on the third run against Rybka2.3.2a. 80
Table 11 Final weight values of black pawns than obstruct the black bishop's movement. 83
Table 12 Average weight values and their standard deviations for run number 31 (generation 0) 91
Table 13 Average weight values and their standard deviations for run number 31 (generation 50) 94
Table 14 Ratings for the human player and our chess engine in a ten-game match. The final result was 9 to 1 for the human player. 95
Table 15 Ranges of the weights for each virual player. 107
Table 16 Values of the weights after the exploration search (shown in the second column) and after the exploitation search (shown in the third column). 110
Table 17 Ratings of the second experiment. 111
Table 18 Ratings of the third experiment. 112
Table 19 Elo rating system 118
Table 20 Some values for the relationship between rating difference and expected score. 119
Table 21 Top ten chess players until October 2012. 122
Table 22 Grandmasters per country until July 2012. 122

"Though I would have liked my chances in a rematch in 1998 if I were better prepared, it was clear then that computer superiority over humans in chess had always been just a matter of time."
Garry Kasparov

1

INTRODUCTION

This chapter describes the problem solved in this doctoral work, the hypotheses and both general and specific objectives to carry it out. It also gives a brief description of the contents of the chapters in this document.

1.1 STATEMENT OF THE PROBLEM

Build a chess engine with a rating of around 2600 points in the ELO system (see Appendix A), by applying artificial intelligence techniques.

1.2 HYPOTHESIS

- The use of artificial intelligence techniques will improve the rating of a chess engine; in particular, the use of evolutionary computation and/or neural networks.
- Evolutionary computation techniques and/or neural networks can adjust the weights of a chess engine; in particular, the material values and positional values of the pieces.

1.3 OBJECTIVES

The general objectives of the doctoral work were:

- Design and implement a chess engine with a rating of around 2600 points.
- Add knowledge to the evaluation function of a chess engine through artificial intelligence techniques using principally evolutionary computation and/or neural networks.

The specific objectives of the doctoral work contemplate to carry out the implementation of the fundamental components of a chess engine. Such components are:

- Board representation
- Move generator
- Search algorithm
- Universal Chess Interface communication protocol
- Techniques for transposition tables
- Use of existing databases in the opening phase of the game

1.4 CONTENTS OF THE DOCUMENT

In Chapter 2, we will give a brief description of the computer chess history. Also in that chapter, we will illustrate basic concepts such as game tree, search tree, 0×88 method, evaluation function and search algorithms such as minimax, negamax, branch-and-bound and alpha-beta. In Chapter 3, we will give a short history of neural networks, as well as a description of their basic model, their different types of activation functions, their advantages and disadvantages and their architecture. Also in that chapter, we will talk about different paradigms of evolutionary computation, with emphasis on evolutionary programming. At the end of that chapter, we will refer to related work for adjusting weights of the evaluation function of a chess engine through artificial intelligence techniques, especially neural networks and/or evolutionary algorithms.

Since manual weights adjustment of the chess engine evaluation function requires a significant amount of time (usually years [21], [6] and [5]) in Chapters 4, 5, and 6, we propose different methods to carry it out automatically. In Chapter 4, we propose an original neural network architecture to obtain the

positional values of chess pieces. Weights adjustment of such neural networks was done through the use of an evolutionary algorithm producing an increase of 433 rating points in our chess engine (from 1745 to 2178 points). In Chapter 5, we present our approach for adjusting weights of the evaluation function. We propose an evolutionary algorithm which has a selection mechanism based on supervised learning through a database of chess grandmaster games. With this approach, we obtained the "theoretical" values of chess pieces. In Chapter 6, we used the Hooke-Jeeves algorithm to continue the weights adjustment of the best virtual player obtained with the evolutionary algorithm in Chapter 5. Using this algorithm as a local search engine, we increased our chess engine rating from 2205 to 2425 points. The use of the Hooke-Jeeves method is an original contribution for adjusting the weights of a chess engine.

In Chapter 7, we will give the general conclusions and some possible paths for future work. In Appendix A, we will show the system employed to rank the players strength in two-player games such as chess. Finally, in Appendix B, is described the UCI communication protocol which is the standard used to establish the communication between the engine and user applications.

"Chess is far too complex to be definitively solved with any technology we can conceive of today. However, our looked-down-upon cousin, checkers, or draughts, suffered this fate quite recently thanks to the work of Jonathan Schaeffer at the University of Alberta and his unbeatable program Chinook."

Garry Kasparov

2

COMPUTER CHESS

This chapter gives a brief summary of computer chess history from its origins to current chess programs. It also provides the basic notions and concepts which we will use throughout the thesis. Concepts such as game tree and search tree are illustrated in Section 2.2. The fundamental components of a chess engine are presented in Section 2.3. The 0×88 board representation method is presented in Section 2.4. The fundamental search algorithms adopted in our search engine (i.e. minimax, negamax, branch-and-bound and alpha-beta) are presented in Section 2.5. Section 2.6 illustrates the concept of evaluation function. Finally, Section 2.7 gives the description of our chess engine architecture.

2.1 COMPUTER CHESS HISTORY

The origins of computer chess date back to the pioneering efforts of Alan Turing and Claude Shannon in the mid and late 1940s. In 1947, Alan Turing [79] designed a program to play chess and, in 1949, Claude Shannon [73], a scientist at Bell Telephone Laboratories, proposed two strategies to implement a chess engine. The first of them, called "Type A", considered all possible moves to a fixed depth of the search tree, and the second, called "Type B", used chess knowledge to explore the main lines to a greater depth. Shannon was the first to estimate that the total number of possible chess games is 10^{120}. Shannon was one of the most distinguished computer scientists in North America and

became to have remarkable contributions in the fields of information theory and computer circuit design. Shannon was also an avid chess player.

Figure 1: Alan Turing

Dates mentioned in the rest of this section were taken from the following references: [69], [74] and the web page http://www.computerhistory.org/chess/index.php from *Computer History Museum*.

Because of its complexity, and the human interest that chess has attracted during many years, this game has been used, since the 1950s, as a benchmark to test a variety of artificial intelligence techniques. In fact, both Turing and Shannon thought that chess was an alternative to achieve the dream that a computer could think.

During the 1950s, chess programs played at a very basic level, but by the 1960s, chess programs could defeat amateur chess players.

In 1952, Alick Glennie, who wrote the first computer compiler, defeated Alan Turing's chess program. He was the first person to beat a computer program at chess.

In 1956, one of the first experiments carried out on the *Univac Computer Maniac I* was to develop a program to play chess. The program used a 6×6 chessboard without bishops and took 12 minutes to search a four moves depth (adding the two bishops would take three hours to search at the same depth). *Maniac I* had a memory of 600 words, it performed $11,000$ operations per second, and had $2,400$ vacuum tubes.

In 1958, Alex Bernstein, an experienced chess player and a programmer at IBM, built the first complete chess program for an IBM 704. This computer

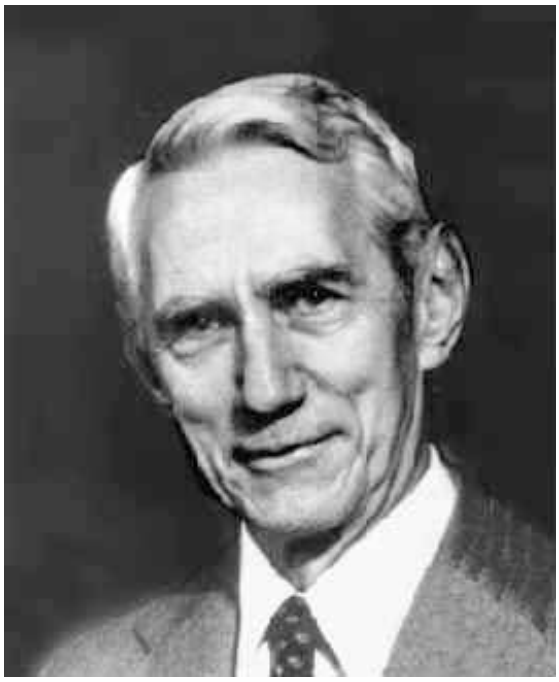

Figure 2: Claude Shannon

could perform 42,000 instructions per second and had a memory of 70 K. The program took eight minutes to search a four moves depth and could play a full chess game, although it could be defeated by novice players.

In 1958, the alpha-beta pruning algorithm [57, 63] for chess was proposed by Allen Newell, Cliff Shaw, and Herbert Simon at Carnegie Mellon University. They developed the NSS (Newell, Shaw, and Simon) chess program, which was different than its predecessors in two aspects: It was the first program written in a high-level language and it used the alpha-beta algorithm for the first time. This program used heuristics that reduced the number of possible moves to explore.

In 1962, the Massachusetts Institute of Technology (MIT) developed its first chess program. It was written by Alan Kotok as part of his B.S. thesis project, assisted by John McCarthy from Stanford University. The program ran on an IBM 7090, looking at 1,100 positions per second. The program could defeat amateur chess players.

In 1965, researches from the Institute for Theoretical and Experimental Physics developed a chess program in Moscow. In 1966, this program began a correspondence match with the Kotok-McCarthy MIT chess program. The match lasted nine months and was won by the soviets, with three wins and one loss.

Figure 3: The Univac Computer *Maniac I*.

In 1967, Richard Greenblatt, a student at MIT, presented a chess program called Mac Hack Six, which was the first to compete respectably against humans in tournament play. It played in several tournaments in Boston and earned a rating of about 1400 points. Greenblatt wrote his program in the assembly language MIDAS for the PDP-6. The program required 16K words and used hash tables [87, 3]. The first tournament victory for Mac Hack Six was against a human having 1510 rating points. Mac Hack Six was also the first to have an opening chess book programmed with it.

In the 1970s, the main chess programs used hash tables which allowed the storage of information about positions that had already been searched. This way, if the same position was reached again, no search was conducted, since the previously generated information would be used in that case. Additionally, other search refinements were also introduced. The most remarkable were: iterative deepening (which searches down to a certain level of the game tree), opening books (which include rules or move sequences that are known to be good to start a game), and endgame databases (which contain move sequences that are known to be good for ending a game, or even solutions to positions with a certain (small) number of pieces). Also, chess programs began to use heuristics and specialized hardware to improve their rating.

In 1970, the Association for Computing Machinery (ACM) organized the first North American Computer Chess Championship, which was held in New York. The chess program Chess 3.0 written by Slate, Atkin and Gorlen at Northwestern University won the tournament. Six chess programs took part

in this event. Table 1 shows the history of the ACM North American Computer Chess Championship from 1970 to 1994. The ACM chess events were cancelled in 1995 because *Deep Blue* was preparing for the first match against world chess champion Garry Kasparov.

Event	City	Participants	Champion
ACM 1970	New York, USA	6	Chess 3.0
ACM 1971	Chicago, USA	8	Chess 3.5
ACM 1972	Boston, USA	8	Chess 3.6
ACM 1973	Atlanta, USA	12	Chess 4.0
ACM 1974	San Diego, USA	12	Ribbit
ACM 1975	Minneapolis, USA	12	Chess 4.4
ACM 1976	Houston, USA	11	Chess 4.5
ACM 1977	Seattle, USA	12	Chess 4.6
ACM 1978	Washington, D.C., USA	12	Belle
ACM 1979	Detroit,USA	12	Chess 4.9
ACM 1980	Nashville, USA	10	Belle
ACM 1981	Los Angeles, USA	16	Belle
ACM 1982	Dallas, USA	14	Belle
ACM 1983 and 4th WCCC	New York, USA	22	Cray Blitz
ACM 1984	San Francisco, USA	14	Cray Blitz
ACM 1985	Denver, USA	10	HiTech
ACM 1986	Dallas, USA	16	Belle
ACM 1987	Dallas, USA	13	ChipTest
ACM 1988	Orlando, USA	12	Deep Thought
ACM 1989	Reno, USA	10	Deep Thought, HiTech
ACM 1990	New York, USA	9	Deep Thought
ACM 1991	Albuquerque, USA	12	Deep Thought II
ACM 1993	Indianapolis, USA	12	Socrates II
ACM 1994	Cape May, USA	10	Deep Thought II

Table 1: History of the ACM North American Computer Chess Championship.

In 1974, the *World Computer Chess Championships* (WCCC) began. Among 13 chess programs that participated in the first tournament, held in Stockolm, Sweden, *Kaissa* was the winner. Table 2 shows the history of the World Computer Chess Championship from 1974 to 2011.

In 1975, Knuth [57] analyzed in detail the alpha-beta pruning algorithm and proposed an improved version that uses a pruning technique which has the advantage of refraining from evaluating some nodes when unnecessary.

Event	City	Participants	Champion
WCCC 1974	Stockholm, Sweden	13	Kaissa
WCCC 1977	Toronto, Canada	16	Chess 4.6
WCCC 1980	Linz, Austria	18	Belle
WCCC 1983	New York, USA	22	Cray Blitz
WCCC 1986	Cologne, West Germany	22	Cray Blitz
WCCC 1989	Edmonton, Canada	24	Deep Thought
WCCC 1992	Madrid, Spain	22	ChessMachine
WCCC 1995	Shatin, Hong Kong, China	24	Fritz
WCCC 1999	Paderborn, Germany	30	Shredder
WCCC 2002	Maastricht, The Netherlands	18	Junior
WCCC 2003	Graz, Austria	16	Shredder
WCCC 2004	Ramat-Gan, Israel	14	Junior
WCCC 2005	Reykjavik, Iceland	12	Zappa
WCCC 2006	Turing, Italy	18	Junior
WCCC 2007	Amsterdam, The Netherlands	11	Zappa
WCCC 2008	Beijing, China	9	Hiarcs
WCCC 2009	Pamplona, Spain	9	Junior, Shredder
WCCC 2010	Kanazawa, Japan	9	Rondo, Thinker
WCCC 2011	Tilburg, The Netherlands	9	Junior

Table 2: History of the World Computer Chess Championship.

In 1977, the *International Computer Chess Association* (ICCA) was founded by computer chess programmers. In this year, Michael Stean became the first grandmaster to lose with a chess program in a blitz game (chess games with five minutes for each player).

Also, in this year, at Bell Laboratories, Ken Thompson and Joe Condon took the brute force approach by developing *Belle*, which was the first computer system to use custom design chips to increase its playing strength. It increased its search speed from 200 positions per second to 160,000 positions per second (eight ply). *Belle* won the North American Computer Chess Championships in 1978, 1980, 1981, and 1982.

During the early 1980s, chess programs based on microprocessors became reachable to a larger audience. However, this technology also made such pro-

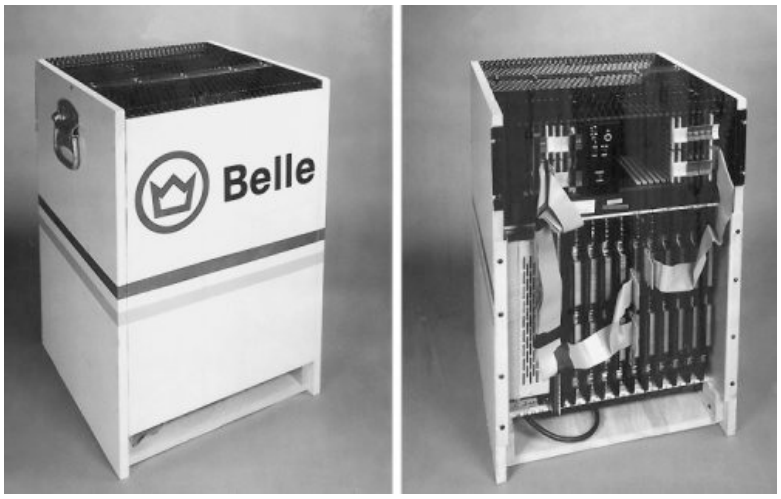

Figure 4: The chess computer *Belle*.

grams very limited due to the small memory capabilities and the slow processors available at that time. During the 1980s, tournaments between chess programs and humans started.

In 1989, two chess computers developed at Carnegie-Mellon University (*Hitech* and *Deep Thought*) were able to defeat a human chessmaster each. In this year, IBM hired Deep Thought team members Feng-Hsiung Hsu, Murray Campbell and Thomas Anantharaman to develop a computer that would beat reigning World Chess Champion Garry Kasparov. Kasparov easily defeated the chess computer *Deep Thought* in both games of a two-game match in 1989.

In 1990, the former chess World Champion Anatoly Karpov lost with the chess program Mephisto in a simultaneous exhibition in Munich. Mephisto also beat grandmasters David Bronstein and Robert Huebner.

In 1992, Kasparov played against the chess program *Fritz 2* in an eleven blitz games in Cologne, Germany. Kasparov won the match with six wins, one draw, and four losses. This was the first time a program defeated a world champion at a blitz game.

In March, 1993, the grandmaster Judit Polgar lost to *Deep Thought* in a 30 minute game.

In 1994, Kasparov lost to *Fritz 3* in Munich in a blitz tournament. The program also defeated the grandmasters Anand, Short, Gelfand, and Kramnik. Kasparov played a second match with *Fritz 3*, and won with four wins, two draws, and no losses. Also, in 1994, at the Intel Speed Chess Grand Priz in

London, Kasparov lost to *Chess Genius 2.95* in a 25 minute game. This defeat eliminated Kasparov from the tournament.

Round	1	2	3	4	5	6	Total
Garry Kasparov	0	1	0.5	0.5	1	1	4
Deep Blue	1	0	0.5	0.5	0	0	2

Tabla 3: Results of the match Kasparov vs Deep Blue (1996).

Figure 5: *Deep Thought's* team.

Round	1	2	3	4	5	6	Total
Garry Kasparov	1	0	0.5	0.5	0.5	0	2.5
Deep Blue	0	1	0.5	0.5	0.5	1	3.5

Tabla 4: Results of the match Kasparov vs Deep Blue (1997).

By 1996, a new improved version of the computer *Deep Thought* was named *Deep Blue*. *Deep Blue* was able to examine 100 million chess positions per second, and played a six-game match versus world chess champion Garry Kasparov. The match was organized by the Association for Computing Machinery to mark the 50th birthday of the first computer. The chief organizer was Monty Newborn, professor of computer science at McGill University. In the

Figure 6: Kasparov vs *Deep Blue*.

first game, *Deep Blue* made history by defeating Kasparov. This was the first time a current world chess champion had ever lost a game using normal time controls. The final result of the match was 4 to 2 in favor of Kasparov (see Table 3). Kasparov won $400,000 US and Deep Blue's team won $100,000 US.

In 1997, *Deep Blue* was used again to play a six-game match against Garry Kasparov. This time, however, Kasparov lost the match (he obtained 2.5 points and *Deep Blue* obtained 3.5 points). This was the first time a computer defeated a reigning world champion in a classical chess match. After almost 50 years of research, the goal of having a computer that was able to defeat the chess world champion had finally been fulfilled. Table 4 shows the match's result. Deep Blue's team won $700,000 US and Kasparov won $400,000 US. IBM estimated that the corporation received $50,000,000 US worth of publicity during the match.

In August 2000, the chess program *Deep Junior* took part in a super grandmaster tournament in Dortmund, Germany. It obtained 50% of the possible points and a performance rating of 2703.

In 2002 a match between the grandmaster Mikhail Gurevich and the chess program *Junior 7* was held in Greece. Junior won with three wins and one draw. In the same year the grandmaster Kramnik drew a match with the chess program *Deep Fritz* with a 4-4 score. Also, a match between Kasparov and the chess engine *Deep Junior* was carried out. The final result was 3 to 3.

In 2003, a match between Kasparov and the chess program *Deep Junior 7* was held in New York, USA. The match ended in a draw. *Deep Junior* took ten years in being programmed by Amir Ban and Shay Bushinksy. It can evaluate three million moves per second and it has a search depth of 15 moves.

The Hydra super-computer was developed by Ali Nasir Mohammed as a project manager. Hydra has 16 Xeons running at 3.06 GHz each, with about 16 GBytes of RAM in the whole system. Hydra explores 200 million positions per second, has a search depth of 18 ply (9 moves by each player) and uses the alpha-beta pruning algorithm. It has about 3000 rating points. In 2004, Hydra defeated grandmaster Evgeny Vladimirov with three wins and one draw. It then defeated former world champion Ruslan Ponomariov (rated at 2710 points) in a two-game match, winning both games. In June, 2005, Hydra beat Michael Adams, who was ranked as the 7th best chess player in the world. Hydra won five games and drew one. Figure 7 shows the Hydra super-computer.

Figure 7: *Hydra* super-computer.

2.2 NOTIONS AND CONCEPTS

Chess is a two-player zero sum game (where the gain of one player is offset by the loss of another player) of perfect information (i.e., all the available information is known by all the players at all times, because each player has access to information on the position of his opponent and his possible moves). In chess, there are two opponents playing and performing moves alternately. On each

turn, the rules define the legal moves. These games start in a specific initial state and end in a position that may be declared as draw, victory or defeat to one side in particular.

2.2.1 *Game tree*

The *state-space* of a game is the number of legal game positions reachable from the initial position of the game. A *game tree* is a representation of the state-space of a game. A *node* in the tree represents a position in the game, an *edge* in the tree represents a move. A *path* is a sequence of edges where each edge shares one node in common with the preceding edge, and the other node in common with the succeeding edge. The root of the tree is a representation of the initial position. A *terminal position* is a position where the rules of the game determine if the result is a win, a draw, or a loss. A *terminal node* represents a terminal position. A *child* of a node is a direct successor of this node. Analogously, the direct predecessor of a node is called the *parent* of the node. A node with at least one successor is called an *interior* node. The root is the only node without a parent. Terminal nodes have no successors. The game stops when a terminal node is reached. Figure 8 shows graphically the description for the root node (node number 1), the interior nodes (nodes number $1, 2, 3, 4$) and the terminal nodes (nodes number $5, 6, \ldots, 13$).

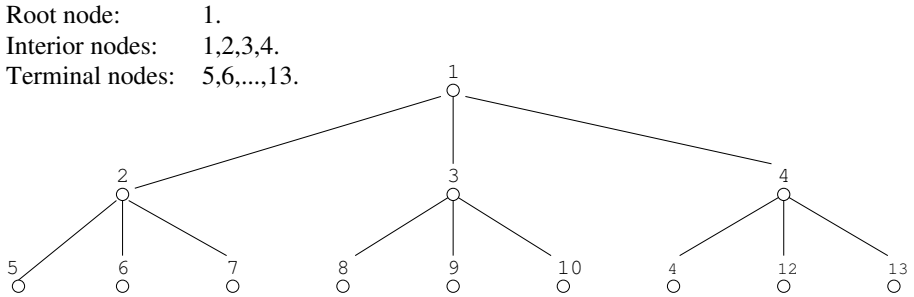

Figure 8: Nodes description in the game tree.

We call the first player MAX and his opponent MIN. Also, we will refer to the positions of the game where MAX moves as MAX positions, and where MIN moves as MIN positions. The trees that represent the game have two types of nodes: nodes MAX (even levels from the root) and nodes MIN (odd levels from the root). We will distinguish the positions where MAX moves with squares, and where MIN moves with circles (see Figure 10).

2.2.2 *Search tree*

Much of the work on search in artificial intelligence deals with trees. The process of searching for a solution of the given problem can be represented by a search tree. While the chess game could be solved in principle, this is not possible within any practical time. Hence, we have to make decisions for a good next move without knowing the game theoretical values of the positions. These decisions usually have to be based on heuristic estimates.

Because in chess, the game tree is too large to be generated completely, this is not feasible in practice. In chess, the game tree consists of roughly 10^{43} nodes [73]. Thus, a *search tree* is generated instead. This search tree is only a part of the game tree. The root represents the position under investigation (normally the initial position), and the remaining nodes in the search tree are generated during the search process. The nodes without children are called *leaves*. Leaves include terminal nodes and nodes which are not yet expanded. A ply is a half move (a move by one of the two players). A path from the root to a leaf is called *variation*. Leaves are evaluated with the *evaluation function*. A *principal variation* is a sequence of moves where both players play optimally.

2.3 FUNDAMENTAL COMPONENTS

A chess engine is a computer program that plays chess. A chess engine receives a chess position as input and calculates the best move depending on the time available. Its main components are:

- Board representation
- Move generation
- Search algorithms
- Evaluation function

It is worth noticing that the graphical user interface (GUI) is not part of a chess engine. The communication between the chess engine and the graphical user interface is through the *Universal Chess Interface* (UCI) protocol, which will be explained in Appendix B. Next, we will describe these components.

2.4 BOARD REPRESENTATION AND MOVE GENERATION

The 0×88 method is composed by an array of integers as shown in Figure 9. The size of this array is $16 \times 8 = 128$, numbered from 0 to 127. It is basically two boards next to each other. The real board is on the left (shown with a thicker line), and the board on the right is an imaginary board that represents illegal moves. The numbers c inside the real board have the characteristic that the fourth and eighth bits are zero and the number $0 \times 88 = 10001000_2$ has the characteristic that the fourth and eighth bits are one. Thus, the operation c AND 0×88 is equal to zero for all squares inside the real board.

The *move generation* of a chess engine is a procedure which generates the possible moves from a given position on the board. This procedure must be an efficient operation because each time the search algorithm visits an internal node in the search tree, it must generate all possible moves. This is achieved through the 0×88 method because with a bitwise operation AND we know if a particular move is valid. If the move generation did not use this method, we would need to perform four arithmetic operations to determine whether the move of a piece had exceeded the end's bottom, top, left or right of the board.

112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Figure 9: Board representation with the 0×88 method.

Algorithm 1 shows the procedure $generateMoves(\texttt{int}\ s, \texttt{int}\ *a)$ which generates the moves of the chess piece defined in the array a from the square s. The second argument of this function must be a pointer to a constant array of integers and must define the immediate moves for each type of chess piece. For example, for the rook, this array is defined by:

$arrayRook = \{-1, 16, 1, -16, 0\}$,

where the values -1, 16, 1, -16 define the immediate moves of the rook to the left, up, right and down, respectively. The value 0 at the end of this array is used to finish the move generation of this piece.

So, if we want to carry out the move generation for a rook on square 20, we will have to invoke this procedure as follows:

$generateMoves(20, arrayRook)$

The procedure $saveMove(s, i)$ in Algorithm 1 saves in the array of moves, the move from the square s to square i.

Algorithm 1 $generateMoves(int\ s, int\ *a)$

```
1: for (;*a;a++) do
2:    int i;
3:    for (i = s + (*a);!(i & 0x88);i+ = *a) do
4:       saveMove(s,i);
5:    end for
6: end for
```

2.5 SEARCH ALGORITHMS

2.5.1 *Minimax*

The *minimax algorithm* is the fundamental algorithm for games between two adversaries. Basically, this algorithm assumes there are two players called MAX and MIN, and assigns a value to every node in a search tree that represents the possible moves of the two players. Terminal nodes are assigned static values that represent the value of the position from MAX's point of view. Non-terminal nodes receive their values recursively using a depth first search algorithm [15]. If a non-terminal node p has MAX to move, then the value of p is the maximum of the values of its successor nodes. Correspondingly, if a non-terminal node p has MIN to move, then the value of p is the minimum of the values of its successor nodes. The objective of the algorithm is to find the principal variation from the root node (initial position), which is given by the sequence of moves where two players play optimally. Figure 10 shows an example of the operation of the minimax algorithm. In this figure, the squares represent the player MAX, the circles represent the player MIN and the thick line shows the principal variation.

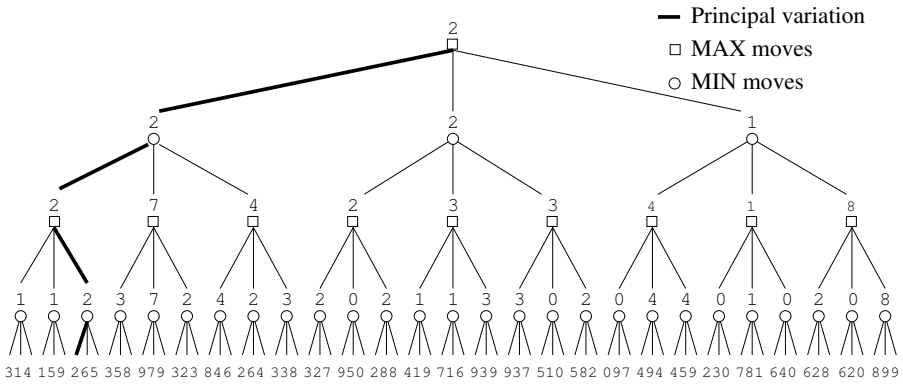

Figure 10: Example of operation of the minimax algorithm.

2.5.2 *Negamax*

The *negamax algorithm* [11] is a more elegant implementation of the minimax algorithm. The problem with the minimax algorithm is that we have two different functions that are essentially doing the exact same thing. Thus, the negamax approach applies the same operator at all levels in the tree. In the negamax procedure, the terminal nodes are assigned static values from the point of view of the side to move. This allows the value of non-terminal positions to be calculated as the maximum of the negatives of the values of the successors. Figure 11 shows an example of operation of the negamax algorithm. Again, the thick line shows the principal variant for the initial position (note that this variant is the same as that obtained with the minimax algorithm).

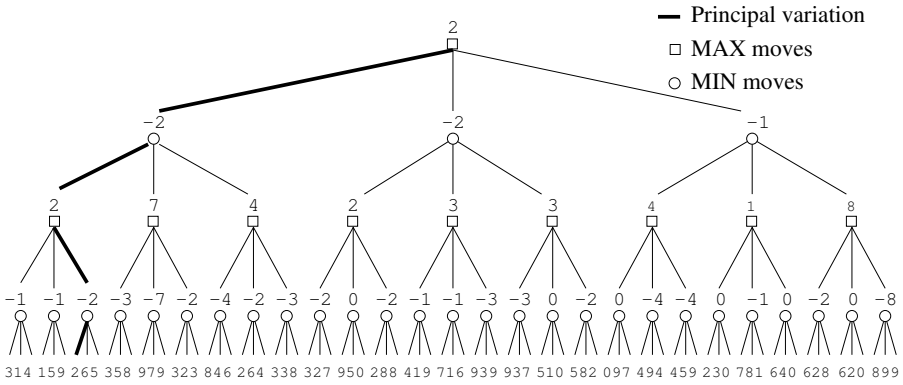

Figure 11: Example of operation of the negamax algorithm.

Algorithm 2 shows the negamax algorithm. This algorithm receives as its input the node p and returns the value assigned to the root node. Line 1

declares the integer variables used in the program. Line 2 determines if the node p is a terminal node, and if so, it determines the value of the position in node p with the evaluation function f(p) (these functions will be discussed in Section 2.6). Line 5 generates all possible moves from node p and line 6 assigns to variable m the value $-\infty$ (∞ denotes a value that is greater than or equal to f(p) for all terminal nodes of the game). Lines 7 to 12 contain the fundamental part of the negamax algorithm. These lines assign to the variable m the maximum of the negatives of the values of the successors of the node p.

Algorithm 2 negamax(p:node)

```
1:  integer m, i, t, w;
2:  if terminal(p) then
3:     return  f(p);
4:  end if
5:  w ← moveGeneration(p); {Determine the successors p_1 ... p_w};
6:  m ← −∞;
7:  for i = 1 a w do
8:     t = -negamax(p_i);
9:     if t > m then
10:       m ← t;
11:    end if
12: end for
13: return  m;
```

2.5.3 *Branch-and-bound algorithm*

Both the minimax and the negamax algorithms are characterized for visiting all the nodes in the search tree. It is possible to improve these brute-force algorithms by using the branch-and-bound algorithm. This procedure ignores moves which are incapable of being better than moves that are already known. Figure 12 shows an example of pruning through the branch-and-bound technique. For example, if $F(P_1) = -1$, then $F(P) >= 1$, and we do not have to know the exact value of $F(P_2)$ if we can deduce that $F(P_2) >= -1$. Thus, if $F(P_{21}) <= 1$, we do not need to bother about exploring any other moves from P_2. Thus, the search for the principal variation in Figure 12 can ignore edges with dotted lines and all successors of node P_2.

The branch-and-bound algorithm is shown in Algorithm 3. Its fundamental difference with respect to the negamax algorithm are lines 12 to 14. These

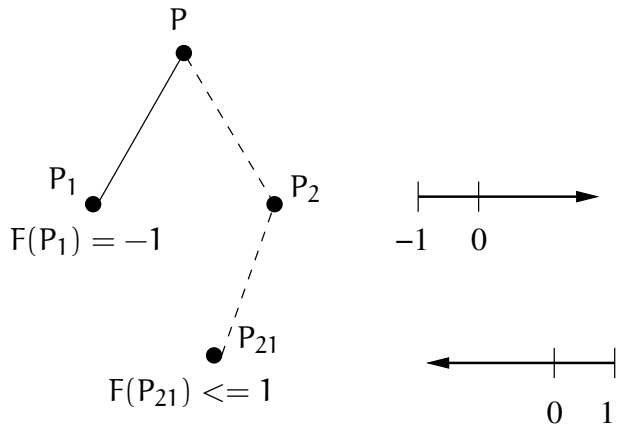

Figure 12: If $F(P_1) = -1$, then $F(P) \geqslant 1$ and we do not have to know the exact value of $F(P_2)$ if we can deduce that $F(P_2) \geqslant -1$. This happens if $F(P_{21}) \leqslant 1$.

lines verify if the right end of the range has been exceeded. If that is the case, a pruning of the tree takes place to avoid visiting the subtree that starts in node p.

In Figure 13, it can be seen how the branch-and-bound algorithm works with the example that we have considered for the minimax and negamax algorithms. This algorithm is called for the first time with the procedure branch-and-bound(p, $-\infty$), where p is the root node. The dotted line shows the edges of the tree that have been pruned and do not need to be visited, and the thick line shows the principal variation of the search tree (note that this variant is the same when using either the minimax or negamax algorithm).

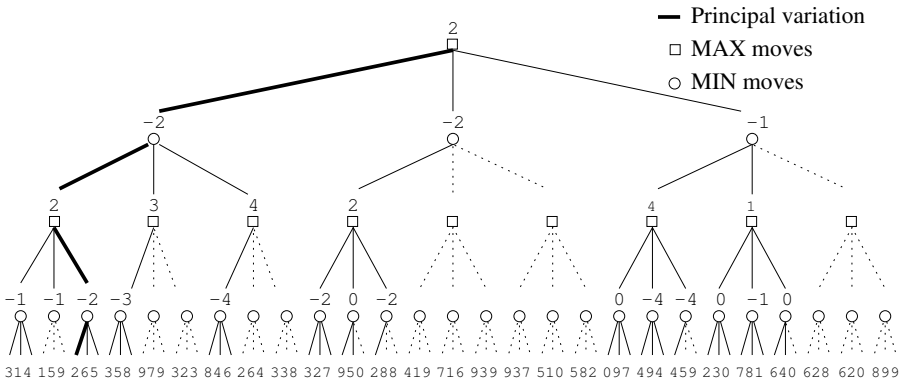

Figure 13: Example of pruning with the branch-and-bound algorithm.

Algorithm 3 branch-and-bound(p:node, bound:integer)

```
1:  integer m, i, t, d;
2:  Determine the successors of the node p : p_1, ..., p_d;
3:  if terminal(p) then
4:     return f(p);
5:  end if
6:  m = −∞;
7:  for i = 1 a d do
8:     t = -branch-and-bound(p_i, −m);
9:     if t > m then
10:       m = t;
11:    end if
12:    if m ⩾ bound then
13:       return m;
14:    end if
15: end for
16: return m;
```

2.5.4 *Alpha-beta pruning*

The branch-and-bound approach can be improved if a lower limit is added to the use of an upper limit. This idea is known as *alpha-beta* pruning [57]. This approach is able to evaluate a game tree at a low cost by ignoring subtrees that can not affect the final value of the root node. This procedure is shown in Algorithm 4. The fundamental difference with respect to the branch-and-bound approach is in line 8, in which the algorithm recursively invokes the alpha-beta pruning scheme, whose input parameters include the current node p, the lower limit alpha and the upper limit beta. The inclusion of both limits allows the search window to be reduced, and in this way is possible to prune further the game tree. Figure 14 shows a search tree pruned with the alpha-beta algorithm which is called for the first time with the procedure alpha-beta(p, $-\infty$, ∞), where p is the root node.

2.5.5 *Quiescence search*

The quiescence search is essentially a variation of the alpha-beta algorithm and is used to extend the search tree to steady positions in which material

Algorithm 4 alpha-beta(p:node, alpha:integer, beta:integer)

```
1: integer m, i, t, d;
2: Determine the successors of the node p: p_1, ..., p_d;
3: if d = 0 then
4:    return f(p);
5: end if
6: m = −∞;
7: for i = 1 a d do
8:    t = -alpha-beta(p_i, −beta, −m);
9:    if t > m then
10:      m ← t;
11:   end if
12:   if m ⩾ beta then
13:      return m;
14:   end if
15: end for
16: return alpha;
```

exchanges, king's checks, and pawns promotion cannot influence the evaluation of a given position. The purpose of this algorithm is to avoid stopping the search for the principal variation in a critical situation. For example, let's consider the chessboard position in Figure 15 and let's assume that the black queen on f6 takes in the next move the white bishop on f4. If at this moment, the search is stopped and the position is evaluated, we might think that the black side has won a bishop. However, considering one more move, it can be seen that the black queen is also lost because it would be captured by the white rook on f1 or by the white queen on g3. Therefore, we need to evaluate the positions until there is no more material exchange, i.e., when we have reached steady positions.

2.5.6 *Iterative deepening*

In chess it is very important to be able to generate a reasonable decision at any time. If the algorithm used has not completed the search when the time limit has been reached, then a catastrophic move may be produced. The idea of iterative deepening is to run the alpha-beta algorithm down to depth 1, 2, ... In this way, it will always be available to obtain a correct answer of the

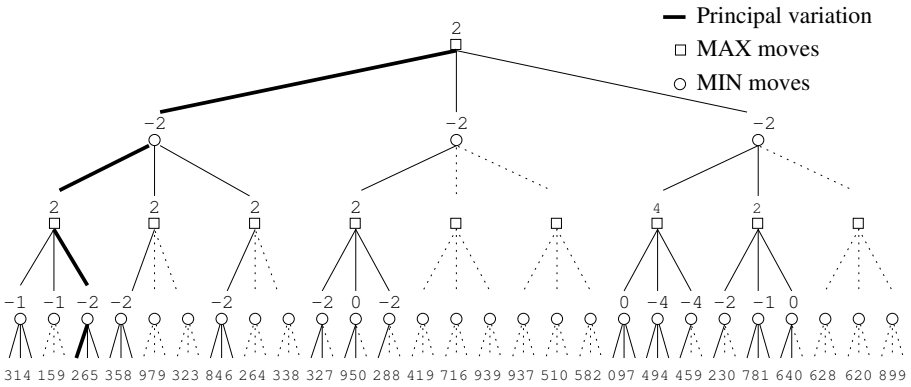

Figure 14: Example of pruning with the alpha-beta algorithm.

search algorithm at any time. At a first glance this technique slows down the search. However, it presents more cuts in the search tree because the moves of the previous search can be sorted. Instead, the chess position values obtained with the evaluation function to depth $d-1$ can be stored in a table so that it is not necessary to calculate them again to depth d (see hash table in Section 2.7, in page 26).

2.6 EVALUATION FUNCTION

As we saw in Section 2.2, it is impossible to build the full search tree due to the complexity of chess games; instead, we decided to construct a game tree which is a representation of the legal game positions reachable from the initial position of the game. Possibly, the leaf nodes of the game tree are non-terminal nodes, so they must be assigned a numeric value through the evaluation function. This function is used to determine (in a heuristic way) the relative value of a position with respect to a particular side. The aim is that the evaluation function reflects the knowledge of the game and it must be efficient to achieve a greater depth in the search tree. In [62], it is considered that for each additional layer of search, a chess engine should increase its strength between 200 and 250 Elo points.

The evaluation function is dependent on the phase of the chess game (opening, middle game or final). Thus, the opening phase could be evaluated through an evaluation function; however it is preferred to replace this phase by opening databases that have been the result of the knowledge generated over centuries on chess games. Also, the final phase of the game with five or less chess pieces is usually based on databases because there are algorithms that can solve all

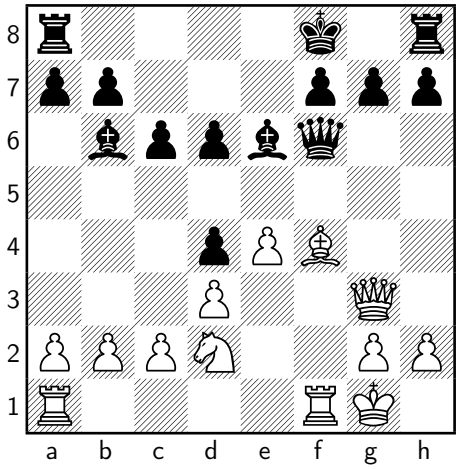

Figure 15: Example diagram.

the finals for these numbers of pieces. However, given the size of the search space of the chess game (10^{43} different chess positions), in the middle phase there are no databases that can define the best move for a particular position, and, therefore, in such cases, an evaluation function is required. In this phase of the game, the main aspects to be considered are:

- **Material balance**. The material values of the chess pieces should be taken into account.
- **Mobility of the pieces**. The number of available moves of the chess pieces should be considered.
- **King safety**. The number of pieces that are defending its king, that are attacking its king, the pawns structure that is defending its king, the castle, etc., should all be considered.
- **Pawn structure**. Doubled pawns, isolated pawns, passed pawns and backward pawns, among others, should be considered.
- **Location of pieces**. For example: rooks on open or semi-open columns, rooks on seventh row, knights on the fifth or sixth row cannot be evicted from their position by an opponent pawn, bishops on open diagonals, trapped pieces, etc.

- **Control board**. The control of the board center, the critical squares in a certain position, the opponent's space, etc. should all be considered.

If the value returned by the evaluation function is in the interval $(-1, 1)$, then the position is of balance and both sides are on equal conditions. If the value returned by the evaluation function is less than or equal to -1, then the black side has a significant advantage as to win the game. Similarly, if the value returned by the evaluation function is greater than or equal to 1, then the white side has a significant advantage as to win the game.

The evaluation function is the most important component of a chess engine. The evaluation function contains weights in arithmetic expressions which encode specific knowledge that constitutes a very valuable source of information for the search engine. If the weights used in the evaluation function are improved, then the chess engine will be better (i.e., it will play better). Developers of commercial chess programs must fine-tune the weights of their evaluation functions using exhaustive test procedures (which may take years), so that they can be improved as much as possible. However, a manual fine-tuning of weights is a difficult and time consuming process, and therefore the need to automate this task. In this work, we propose two methods for adjusting the weights of the evaluation function based on artificial intelligence techniques. The first method is illustrated in Chapter 4, and the second is shown in Chapters 5 and 6.

2.7 OUR CHESS ENGINE

In Figure 16, our chess engine architecture is shown. The procedure *Search* contains the implementation of the alpha-beta algorithm (Section 2.5.4) or quiescence search (Section 2.5.5). This procedure returns the value `val` of the current position to the procedure *Iterative deepening* (if this procedure contains the alpha-beta algorithm, then it also returns the principal variation of the game tree). This procedure is invoked with parameters `alpha` (lower limit), `beta` (upper limit) and (`depth`) (Section 2.5.4).

The procedure *Search* also invokes the procedure *Evaluation function* which returns the numeric value `val` of the current position `pos` (Section 2.6).

The procedure *Move Generation* is invoked by the procedure *Search* and it generates the moves of the chess pieces (Section 2.4) from the current position `pos`. This procedure receives the parameters `pos` and `type`, where the last parameter defines the type of moves to generate. If this procedure is called by the alpha-beta algorithm, then it generates all possible moves, but if it is called

by the quiescence search, then it generates only the special moves (exchange of material, checks to the king, etc.).

The procedure *Hash table* is used to store positions that have been evaluated before, so that they do not have to be calculated again. Each position is identified by a key, which serves as an index in the array of the hash table. Each position in this table contains the value of the respective position obtained with the evaluation function. In chess, there are two different reasons for which the same chess position may occur several times. The first is transposition, in which the same chess position can be reached by different sequences of moves. The second is due to the iterative deepening algorithm, in which the same chess position is repeated at different depths of search (Section 2.5.6). This procedure receives the current position pos as a parameter and returns the array index in the hash table or -1 if the position pos was not found.

Finally, the procedure *Iterative deepening* communicates with the *Arena* graphical user interface[1] through the UCI communication protocol (see Appendix B).

2.8 FINAL REMARKS OF THIS CHAPTER

The principal goal of this work is to develop a chess engine with automatic tuning of weights based on artificial intelligence techniques. In this chapter we presented the basic notions and components of a chess engine. Also, we showed the architecture of our chess engine. These concepts are necessary to describe our contributions in Chapters 4, 5 and 6.

1 *Arena* is available at http://www.playwitharena.com/

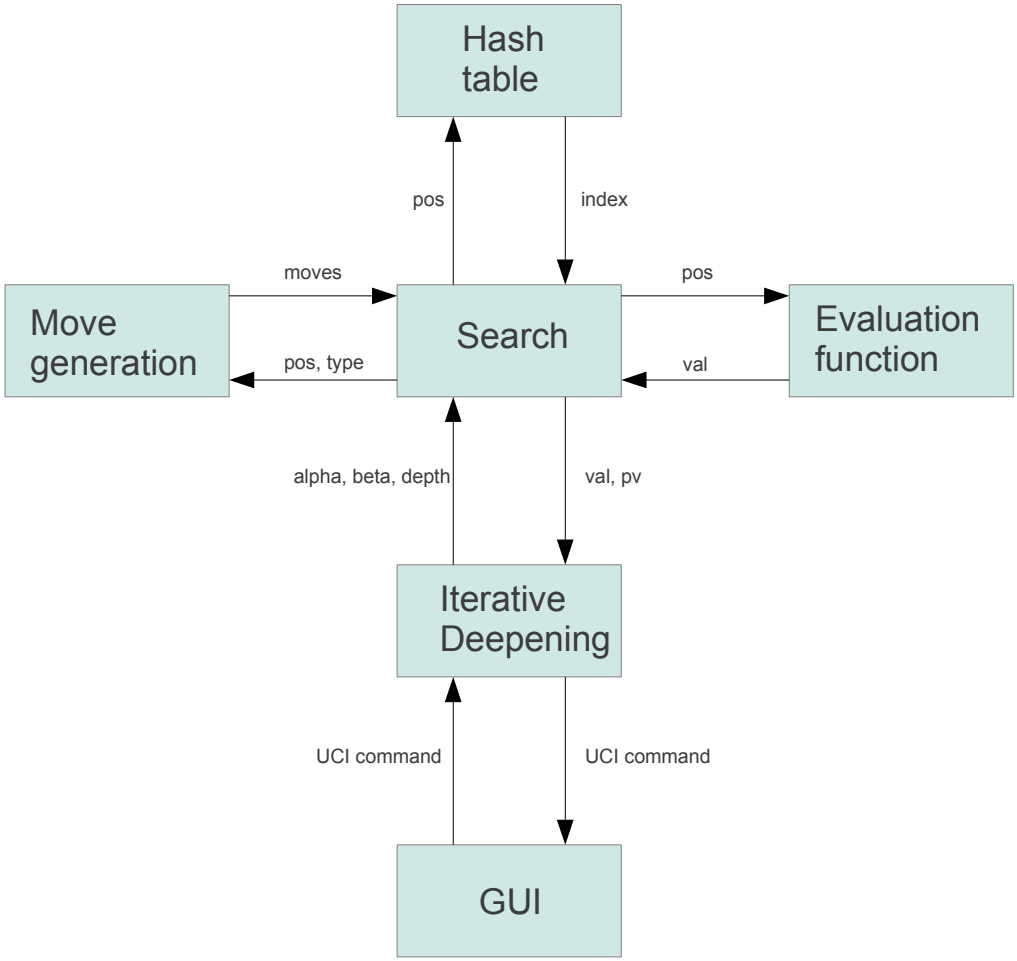

Figure 16: Architecture of our chess engine.

"It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change."

Charles Darwin

3

SOFT COMPUTING IN CHESS

In this chapter, we will provide a short history of neural networks (Section 3.1.1), as well as a description of the basic model of an artificial neural network (Section 3.1.2), the different types of activation functions for the output of a neuron (Section 3.1.3), the advantages and disadvantages of a neural network (Section 3.1.4), the architecture of a neural network (Section 3.1.5), and the different types of neural networks available based on their learning process (Section 3.1.6).

Also, this chapter provides a basic description of an evolutionary algorithm (Section 3.2.1) and its components (Section 3.2.2). It also provides a comparison between evolutionary algorithms and traditional search techniques (Section 3.2.3). Finally, it presents the three main evolutionary computation paradigms (Section 3.2.4), as well as some final remarks.

3.1 ARTIFICIAL NEURAL NETWORKS

3.1.1 *A short history of neural networks*

In 1943, Warren McCulloch, a psychiatrist and neuroanatomist, and Walter Pitts, a logician, presented the first model of an artificial neuron [64]. They formally defined the neuron as a binary machine with multiple inputs and outputs, and they used the neural networks to model logical operators. Their work is still a cornerstone in the theory of neural networks.

In 1949, the physiologist Donald Hebb outlined in his book *The Organization of Behavior* [47] the first rule for self-organized learning. He assumed that learning is located in the synapses or connections among neurons. Donald Hebb's book was immensely influential among psychologists, but, unfortunately, it had little impact in the engineering community.
In 1951, Marvin Minsky and Dean Edmonds built the first neural network machine, called "Sharc". It was composed of a network of 40 artificial neurons that imitated the brain of a rat.
In 1954, Marvin Minsky wrote his Ph.D. thesis which was entitled "Theory of Neural-Analog Reinforcement Systems and its Application to the Brain-Model Problem".
In 1958, Frank Rosenblatt introduced the concept of Perceptron which is a more sophisticated model of the neuron [27]. This is the first model for learning with a teacher (supervised learning). The Perceptron was used for pattern classification, which must be linearly separable (i.e., patterns that lie on opposite sides of a hyperplane). Basically, it consists of a single neuron with adjustable synaptic weights and bias (see Section 3.1.2).
In 1959, Bernard Widrow and Marcian Hoff developed the models Adaline (ADAptive LINear Element) and Madaline (Multiple Adaline), which constitute the first applications of real networks to real world problems. Their basic difference is that the Adaline net is limited to only one output neuron, while Madaline can have many.
In 1960, Bernard Widrow and Marcian Hoff introduced the delta rule (also known as Widrow-Hoof rule or least mean square) which was used to train Adaline [27].
In 1967, Marvin Minsky published the book *Computation: Finite and Infinite Machines* [67]. This book extends the results of Warren McCulloch and Walter Pitts and put them in the context of automata theory and the theory of computation. Also in this year, Shun-ichi Amari used the stochastic gradient method for adaptive pattern classification [52].
In 1969, the Perceptron was strongly criticized by Marvin Minsky and Seymour Papert because it was only capable of solving linear problems. They provided a mathematical proof of the fundamental limits on which a single-layer Perceptron can compute. They stated that there was no reason to assume that any of the limitations of the single-layer Perceptrons could be overcome in the multilayer version.

From the viewpoint of engineering, the decade of 1970s was an inactive period for the development of neural networks. However, in 1973 the *self-organizing maps* were introduced by Christoph von der Malsburg [58].
In 1975, the first multilayer neural network was developed [27].
In 1977, the emerging models of associative memories were introduced by James A. Anderson.
In 1982, Teuvo Kohonen published his self-organizing maps [58] which were different in some aspects from the earlier work done by Willshaw and Christoph von der Malsburg [59]. Also, in this same year, Hopfield built a network with symmetric synaptic connections and multiple feedback loops which was initialized with random weight values which eventually reached a final state of stability.
In 1983, Andrew G. Barto, Richard S. Sutton, and James A. Anderson published their seminal paper on *reinforcement learning* [76] (although Marvin Minsky used this concept for the first time in his Ph.D. thesis from 1954).
In 1986, David E. Rumelhart, Geoffrey E. Hinton and Ronald J. Williams proposed the backpropagation algorithm [46].

3.1.2 *Basic concepts of artificial neural networks*

An artificial neural network is a mathematical model that aims to simulate the behavior of the brain. This model is based on studies on the essential characteristics of neurons and their connections. However, although the model turns out to be a very distant approach to biological neurons, it has proved to be very interesting because of its ability to learn and associate patterns which are difficult to solve through traditional programming. Over the years, this model of the neurons has become more complicated in their structure and more flexible in their use.

Artificial neural networks (ANNs) are used in many cases as *black boxes* where a certain input should produce the desired output but we are unaware of the way in which such an output is generated. In general, we are interested in mapping and n-dimensional real-numbers input $(x_1, x_2, \ldots, x_n)$ to an m-dimensional real-numbers output $(y_1, y_2, \ldots, y_m)$. Thus, a neural network works as a machine capable of mapping a function $F : \mathfrak{R}^n \to \mathfrak{R}^m$.

A neuron is a processing unit that is fundamental to the operation of a neural network. Figure 17 shows the model of a neuron. Its three basic elements are:

1. A set of *synapses* w_{kj} (synaptic weights). Specifically, a signal x_j at the input of synapse j connected to neuron k is multiplied by the synaptic weight w_{kj}.
2. An *adder*. It is responsible for adding the input signals multiplied by its respective synaptic weight.
3. An *activation function*. The role of the activation function is to limit the amplitude of the output of a neuron, and it regularly ranges from 0 to $+1$ (see Section 3.1.3).

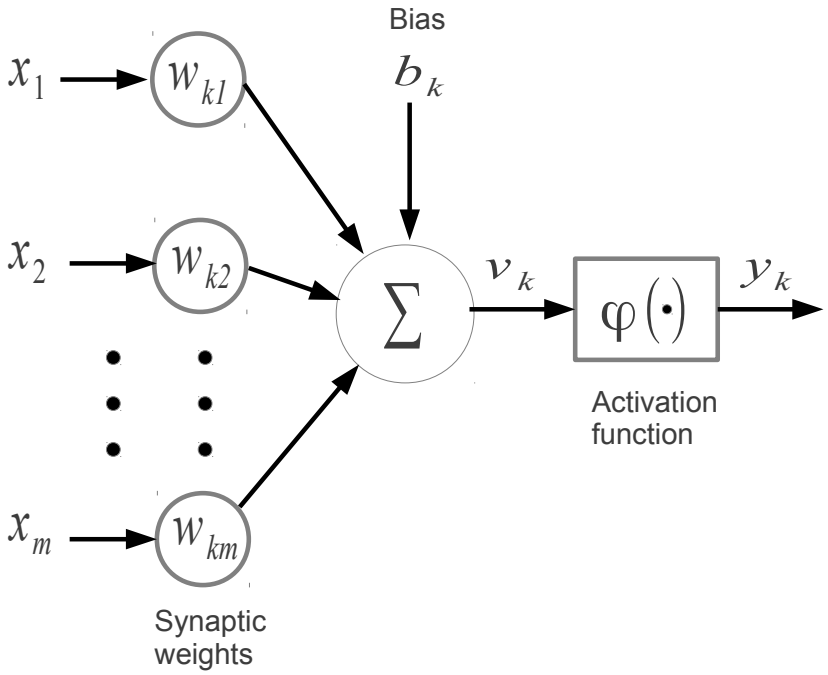

Figure 17: Model of a neuron.

The neural model of Figure 17 also includes an externally applied *bias* b_k. This term is necessary to set an activation threshold of the neuron. We can describe the neuron k in Figure 17 by the following equations:

$$u_k = \sum_{j=1}^{m} w_{kj} x_j \tag{1}$$

and

$$y_k = \varphi(u_k + b_k) \tag{2}$$

Where $x_1, x_2, \ldots x_m$ are the input signals; $w_{k1}, w_{k2}, \ldots, w_{km}$ are the respective synaptic weights; u_k is the linear combined output due to the input signals; b_k is the bias; $\varphi(\cdot)$ is the activation function; and y_k is the output signal of the neuron. The term b_k has the effect of displacing the line given by equation (1) because the value to the output of the neuron k is given by:

$$v_k = u_k + b_k \tag{3}$$

3.1.3 *Activation function types*

The activation function, denoted by $\varphi(v)$, defines the output of a neuron in terms of v. The three different types of activation functions mainly used in neural networks are:

1. **Threshold function**. This type of activation function is defined by the equation:

$$\varphi(v) = \begin{cases} 1 & \text{if } v \geqslant 0 \\ 0 & \text{if } v < 0 \end{cases} \tag{4}$$

2. **Piecewise linear function**. This type of activation function is defined by the equation:

$$\varphi(v) = \begin{cases} 1 & \text{if } v \geqslant +\frac{1}{2} \\ v & \text{if } -\frac{1}{2} < v < +\frac{1}{2} \\ 0 & \text{if } v \leqslant -\frac{1}{2} \end{cases} \tag{5}$$

3. **Sigmoid function**. This is the activation function most commonly used in neural networks. An example of sigmoid function is the logistic function, defined as follows:

$$\varphi(v) = \frac{1}{1 + exp(-\alpha v)} \tag{6}$$

where the constant α is a user-defined parameter.

3.1.4 *Advantages and disadvantages of neural networks*

The main advantages of neural networks are the following [61], [84]:

- Neural networks have demonstrated capability to map and/or approximate any real non-linear continuous function. In fact, the proof of Hecht-Nielsen [48] establishes that any function can be approximated by a three-layer neural network.
- Neural networks can handle incomplete, missing or noisy data.
- Neural networks do not require prior knowledge about the distribution and/or mapping of the data.
- Neural networks can automatically adjust their structure (number of neurons or connections).
- Neural networks are fault tolerant, because the number of connections provides much redundancy, since each neuron acts independently of all others and each neuron relies only on local information.
- Neural networks can be implemented in parallel hardware, increasing the speed of the learning process.
- Neural networks can be highly automated, minimizing human involvement.

Among the different disadvantages of neural networks we have the following [61], [84]:

- The selection of the neural network topology and its parameters lacks theoretical background. An alternative to adjust the structure of the neural network is through intuition or a "trial and error" process [8]. There is still much work to do, but evolutionary algorithms are a promising area for developing automated methods to find optimal topologies. For some work in this area we can see [23], [18] and [55].
- There is no explicit set of rules to select a learning algorithm in neural networks [8].
- Neural networks are too dependent on the quality and the number of available data. In [56] is investigated the effect of data quality on neural network models.
- Neural networks can get stuck in local minima. In [41], are given examples of stagnation of neural networks that use the backpropagation algorithm.

3.1.5 *Neural networks architecture*

The structure of the artificial neural networks is closely related to their learning capabilities. In general, we may identify three main different classes of network architectures:

1. **Feedforward networks with a single layer of neurons**.
   In this type of networks, the neurons are organized in layers. In the simplest form, the network has an *input layer* of source nodes and an *output layer* of neurons which carry out the arithmetic computation. Figure 18 shows a neural network with four nodes in both the input and output layers. This is a network of a single-layer referring to the output layer because no computation is performed in the input layer. In this figure the neurons are denoted with circles and the input nodes with squares. The arrows in the figure show the flow of information which is carried out in one direction (left to right). The difference between node and neuron lies in the fact that the first does not carry out arithmetic computation and the second does.
2. **Multilayer feedforward networks**.
   This type of neural network is identified by the presence of several hidden layers in the neural network whose purpose is to carry out a more

robust learning of the input patterns. Figure 19 shows the architecture of a neural network with four nodes in the input layer, four nodes in the hidden layer and two nodes in the output layer. This neural network is *fully connected* because every node in each layer is connected to every other node in the adjacent forward layer. Also, in this figure, the neurons are denoted with circles and the input nodes with squares, and the flow of information is carried out in one direction (left to right).

3. **Recurrent networks**.
   A recurrent network is a neural network with feedback (closed loop) connections [27]. Figure 20 shows a recurrent network with a single layer of neurons with each neuron feeding its output signal back to the inputs of all the others neurons. In this type of networks is important to incorporate time-delay elements to decide the time at which the information will be fed back.

3.1.6 *Learning process*

Basically, there are two categories of learning within a neural network: learning with a teacher and learning without a teacher. Also, the last type of learning may be sub-categorized in reinforcement learning and unsupervised learning [46]. Each of these types of learning are described below.

1. *Learning with a teacher*.
   This type of learning is also referred to as *supervised learning*.

   In this case, the learning process takes place under the tutelage of a teacher. The network is trained with examples of input vectors and desired target vectors (output patterns), and the objective is to adjust the synaptic weights according to a learning algorithm.

   The most successful and widely used supervised learning procedure for multilayer feedforward networks is called backpropagation algorithm [46]. Girosi and Poggio [35] showed that neural networks with at least one hidden layer are capable of approximating any continuous function. But, Judd [53] showed that the learning problem in neural networks is NP-complete. And although the training time can be very large, any learning procedure which is based on a descent method has no guarantee of converging to the optimal solution (it could converge to a local minimum).

Unfortunately, supervised learning algorithms become unacceptably slow as the size of the network increases when having many hidden layers. However, in practice, backpropagation learning has been successfully applied to several difficult problems [12].

2. *Learning without a teacher*.
In this case, there is no teacher to supervise the learning process. In this type of learning, two subcategories are identified:

   a) **Reinforcement learning**. This is a form of semi-supervised learning in which the learning of an input-output mapping is performed through continued interaction with the environment. Reinforcement learning is different from supervised learning. In this case, the supervisor only tells the neural network if it succeeds or fails in its response to a given input pattern. In this method, each input produces a reinforcement in the weights of the neural network in such a way that improves the production of the desired output. In interactive problems it is often impractical to obtain examples of desired behavior that are both correct and representative of all the situations in which the agent has to act. In this case, a supervisory reward signal is provided which tells the network when it is doing the right thing. Hebbian learning is an example of reinforcement learning [46].

   b) **Unsupervised learning (self-organized learning)**. In this type of learning there is no external teacher or critic to oversee the learning process. In this case, a sequence of input vectors is provided, but no target vectors are specified. The neural network adjusts the weights so that the most similar input vectors are assigned to the same output.

Finally, it should be noted that while the neural networks were initially designed to act as classifiers, they have subsequently been used as optimizers as well. In the latter form, given a fixed topology, specifying the weights of a neural network can be seen as an optimization process with the goal of finding a set of weights that minimizes the network's error on the training set. In fact, in Chapter 4 we use a neural network architecture as an optimizer.

3.2 EVOLUTIONARY ALGORITHMS

3.2.1 *A short review of evolutionary algorithms*

Evolutionary Computation (EC) encompasses a set of bio-inspired techniques based on "the survival of the fittest" which constitutes the basic principle of Neo-Darwinian's theory of natural evolution in which natural selection plays the main role (those individuals that are best adapted to their environment have more opportunities to compete for resources and reproduce [16]).

Evolutionary computation is based on the Neo-Darwinian theory of evolution

The Neo-Darwinian paradigm is a combination of the evolutionary theory of Charles Darwin, the selectionism of Weismann and the inheritance laws of Mendel [29]. Based on this idea, the history of all life in our planet can be explained by the following processes [28]:

Reproduction, mutation, competition and selection are the fundamental processes of the Neo-Darwinian paradigm

1. **Reproduction**. A way to create new individuals from their ancestors. It is an obvious feature of all forms of life in our planet, because without such a mechanism, life itself would have no way of occurring. Reproduction is accomplished through the transfer of an individual's genetic program to its progeny.

2. **Mutation**. It refers to small changes in the genetic material of individuals due to replication errors during the transfer of information of individuals. A mutation is beneficial to an organism if it produces a fitness increase in its adaptation to the environment.

3. **Competition**. This mechanism is a consequence of expanding populations in a finite resource space.

4. **Selection**. In an environment that can only host a limited number of individuals, only the organisms that compete most effectively for resources can survive and reproduce.

In general, to simulate the evolutionary process in a computer we require [66]:

- To encode the structures that will be replicated. Each solution is known as an "individual" and a set of individuals is called a "population".
- Operations to affect the individual's traits.
- A fitness or evaluation function.
- A selection mechanism.

3.2.2 *Components of an evolutionary algorithm*

The most important components of an evolutionary algorithm are [24]:

- Representation.
- Fitness or evaluation function.
- Population.
- Parents selection mechanism.
- Variation operators (recombination and mutation).
- Survivor selection mechanism.

Representation

The evolutionary algorithm works on individuals that represent potential solutions of the problem to be solved. Each individual is represented in a data structure known as *chromosome*. There are two levels of representation used in evolutionary computation:

1. *Phenotype*. It is a possible solution within the original problem context. It is the result of decoding the values of the chromosome into the values of the variables of the problem to be solved. A phenotype is also known as a *candidate solution*.
2. *Genotype*. It is the encoding represented by the chromosome.

The *representation* includes the mapping from phenotypes to their corresponding genotypes and vice-versa. The most used representations in evolutionary algorithms are:

- Integer [14].
- Real [28].
- Binary [49].
- Binary with gray codes [86].
- Trees [60].

Fitness or evaluation function

Evolutionary algorithms are search algorithms that have been widely used to solve optimization problems, and their good performance relies both on their stochastic nature and on an appropriate transformation of the objective function of the problem into a fitness function. The *objective function* is defined in the original context of the problem and the *fitness or evaluation function* is a transformation of the given objective function. The evaluation function assigns a fitness value to each individual in the population. This value measures how well each individual resolves the problem in question with respect to other individuals in the population.

Population

Evolutionary computation techniques work on a population composed of a set of candidate solutions to the problem that we are solving. A population is a multiset of genotypes [24] (multiset is a set where multiple copies of an element are possible). In an evolutionary process, individuals are static objects that do not adapt; it is the population itself the one that changes.

Parent selection

A fundamental part in the operation of an evolutionary algorithm is the process of selecting candidates (parents) to be reproduced. This is typically a stochastic process. Thus, individuals with a higher fitness have a greater opportunity of becoming parents than those with low fitness. Nevertheless, individuals with low fitness are often given a small but non-zero probability of being selected as parents (otherwise, the algorithm could get trapped in a local optimum).

A taxonomy of selection techniques, used with evolutionary algorithms is the following:

1. *Proportional selection*. Originally proposed by Holland [49]. In this technique individuals are chosen according to their fitness contribution. Four main types of approaches are normally considered in this case [39]:
    - Roulette wheel [2].
    - Stochastic remainder. There are two variants with replacement [4] and without replacement [9].
    - Stochastic universal sampling [2].

- Deterministic sampling [22].

2. *Tournament selection*. It is based on direct comparisons of individuals fitness values. There are two versions: deterministic and probabilistic. In the deterministic version, the fittest individual always wins. In the probabilistic version, the fittest individual wins with a certain probability (in some cases the fittest solution will not be selected) [38].

3. *Steady state selection*. Proposed by Whitley [85]. This technique is used in non generational genetic algorithms in which only a few individuals are replaced at each generation. It is particularly suitable when individuals solve the problem in a collective way.

Variation operators (recombination and mutation)

The most widely used evolutionary operators are: mutation and recombination.

Mutation is an stochastic unary operator (operates only on an individual). A mutation operator makes a small change on a single element of a genotype and delivers a modified solution. The mutation operator prevents the stagnation of the search process in a local minima due to the fact that introduces random diversity that normally exceeds the capabilities of the recombination operator.

Recombination or *crossover* is an operator that combines characteristics of two or more individuals with the idea of getting descendants that are better adapted to survive. Recombination and mutation are used to generate new solutions (called *offspring*) which lead the search to the desired regions of the search space.

Survivor selection

It is similar to parent selection, but it is applied in a different phase of the evolutionary process. This mechanism is invoked after creating the offspring, and its purpose is to decide which individuals will pass to the next generation. In survivor selection, for example, when ranking the unified multi-set of parents and offspring and selecting the top segment can be selected (as in our proposed works of Chapters 4, 5, and 6).

The main components of an evolutionary algorithm are shown in Algorithm 5.

Algorithm 5 The general scheme of an evolutionary algorithm

```
1: INITIALIZE population with randomly generated solutions;
2: EVALUATE each individual;
3: while stop condition is not reached do
4:    SELECT parents for reproduction;
5:    RECOMBINE parents that have been selected;
6:    MUTATE the resulting offspring;
7:    EVALUATE new candidates;
8:    COPY the best individual to the next generation (elitism);
9: end while
```

3.2.3 *Evolutionary algorithms versus mathematical programming techniques*

Among the main differences between evolutionary techniques and mathematical programming techniques we have the following:

- Evolutionary techniques using a population of potential solutions instead of a single individual, making them less susceptible to being trapped in local minima/maxima.
- Evolutionary techniques do not require specific knowledge (e.g., gradient information) about the problem they are trying to solve. However, if such knowledge is available, it can be easily incorporated.
- Evolutionary techniques use probabilistic operators while mathematical programming techniques use deterministic operators.

Among the different advantages of the EAs with respect to mathematical programming techniques, we have the following:

- EAs are conceptually very simple.
- EAs have a wide applicability.
- EAs can easily exploit parallel architectures.
- EAs can usually adapt their own parameters.
- EAs have the potential to incorporate domain knowledge and can cooperate with other search/optimization techniques.

- EAs are robust to dynamic changes.
- EAs have been found to have a better performance than mathematical programming techniques in many real-world problems.

3.2.4 *Evolutionary computation paradigms*

Traditionally, evolutionary algorithms have been grouped into three main paradigms:

- Evolutionary programming.
- Evolution strategies.
- Genetic algorithms.

It is worth mentioning that genetic programming [60] is not considered a paradigm of evolutionary computation because it is a specialization of genetic algorithms with representation based on trees. Next, we will briefly explain these three paradigms.

3.2.4.1 *Evolutionary programming*

Evolutionary programming (EP) was proposed by Lawrence J. Fogel [34] in 1960 emphasizing the behavioral links between parents and offspring. Intelligence in this technique can be seen as an adaptive behavior. This approach emphasizes the interactions between parents and offspring.

In evolutionary programming, there is no crossover operator because this technique simulates the evolution at the species level (two different species can not be recombined, e.g. it is not possible to recombine a dog with a cat). EP uses probabilistic selection (usually stochastic tournaments). The only evolutionary operator in this case is mutation, and it operates at a phenotype level, so it requires no encoding of solutions. In the original proposal of EP, self-adaptation of the mutation parameters was not considered, but such a mechanism was incorporated later on.

The basic evolutionary programming algorithm is shown in Algorithm 6.

Algorithm 6 Pseudo-code of evolutionary programming

```
1: Generate randomly an initial population of solutions;
2: Calculate the fitness of the initial population;
3: repeat
4:    Apply mutation to the entire population;
5:    Evaluate each offspring;
6:    Select (using a stochastic tournament) the individuals of the next gene-
      ration;
7: until stop condition is reached
```

In this technique, each individual participates in a predefined number of tournaments and the individuals with more wins will have a higher probability of being part of the population in the next generation.

Some applications of evolutionary programming are:

- Games.
- Forecasting.
- Automatic control.
- Traveling salesperson problem.
- Design and training of neural networks.
- Pattern recognition.
- Routing.

3.2.4.2 *Evolution strategies*

Evolution strategies (ES) were developed in Germany in the mid-1960s by Ingo Rechenberg, Hans-Paul Schwefel and Paul Bienert [72]. Their motivation was to solve complex hydrodynamical problems.

In evolution strategies, mutation is the main operator and recombination (which was not used in its original version) is a secondary operator. In this case, evolution is simulated at an individual level. Recombination can be either sexual (acts on two individuals randomly chosen from the population of parents) or panmictic (more than two parents participate in the offspring generation process). In ES, mutation is used with random numbers generated from a Gaussian distribution. In this paradigm, evolution takes place at a phenotype

level (i.e., no encoding is required) and the selection process is deterministic and extinctive (the worst individuals have zero probability of survival).

The original version $(1+1)-\text{EE}$ used a single parent, which generated a single offspring. If this offspring was better than its parent, it would replace it; otherwise, the offspring would be eliminated. In this version, a new individual is generated using the following equation:

$$x^{t+1} = x^t + N(0, \sigma) \tag{7}$$

where t denotes the current iteration, and $N(0, \sigma)$ is a vector of Gaussian numbers with a mean of zero and a standard deviation of σ.

Rechenberg introduced the concept of population, and proposed an evolution strategy called $(\mu+1)-\text{EE}$, in which there are μ parents which generate a single offspring. Such an offspring is meant to replace the worst parent in the population.

Other models of evolution strategies that have been proposed include $(\mu+\lambda)-\text{EEs}$ and $(\mu,\lambda)-\text{EEs}$. In both models, μ parents generate λ offspring. In $(\mu,\lambda)-\text{EEs}$, the μ best individuals are selected from the offspring, whereas in the $(\mu+\lambda)-\text{EEs}$, the μ best individuals are selected from the union of parents and offspring.

It is worth mentioning that ES evolve not only the variables of the problem, but also the standard deviation which is one of the parameters of the technique. This mechanism is known as "self-adaptation".

The basic algorithm of an evolution strategy is shown in Algorithm 7.

Algorithm 7 Pseudocode of an evolution strategy

```
1: Generate randomly an initial population of solutions;
2: Calculate the fitness of the initial population;
3: repeat
4:     Apply mutation to each offspring based on the success of the previous
       mutations.
5:     Evaluate each offspring;
6:     Select the best individuals for the next generation;
7: until stop condition is reached
```

ES have been applied to solve problems like:

- Network routing.

Algorithm 8 Pseudocode of a genetic algorithm

```
1: Generate randomly an initial population of solutions;
2: Calculate the fitness of the initial population;
3: repeat
4:    Select two parents to create two offspring using crossover;
5:    Apply mutation to each offspring;
6:    Evaluate the mutated offspring;
7:    Select survivors;
8: until stop condition is reached
```

- Applications in Biochemistry.
- Engineering design.
- Optics
- Magnetism.

3.2.4.3 *Genetic algorithms*

Genetic algorithms (GAs) originally called "reproductive plans", were introduced by John H. Holland [49] in the early 1960s. The main interest of Holland was to study natural adaptation in order to apply it to the machine learning. Nowadays, genetic algorithms are the most popular type of evolutionary algorithm.

Goldberg [37] gives a definition of a genetic algorithm:

> Genetic Algorithms are search algorithms based on the mechanisms of natural selection and natural genetics. They combine survival of the fittest among string structures with a structured yet randomized information exchange to form a search algorithm with some of the innovative flair of human search.

The basic genetic algorithm is presented in Algorithm 8 [10].

A genetic algorithm works at a genotype level (because the decision variables have to be encoded) where crossover is its principal operator because a GA simulates evolution at an individual level. Mutation is a secondary operator, and is used to add a new source of diversity to the search process, in order to avoid stagnation. Genetic algorithms use probabilistic selection and they typically adopt binary encoding. It is important to mention that genetic

algorithms require elitism (i.e., the best individual at each generation must be copied to the next population without any changes) in order to guarantee convergence (theoretically) to the global optimum. Although some important theoretical work has been developed around genetic algorithms in recent years [1], its main theoretical foundation lied, for many years, on Holland's schema theorem [49].

Some possible applications of genetic algorithms are [10]:

- Machine learning.
- Optimization (numerical and combinatorial).
- Pattern recognition.
- Query optimization in databases.
- Grammar generation.
- Robot motion planning.
- Prediction.

The main characteristics of the three paradigms that conform evolutionary computation are shown in Table 5.

3.3 DIFFERENTIAL EVOLUTION

Differential Evolution (DE) [17], [70] is a real parameter optimization algorithm. DE searches for a global optimum point in a D-dimensional real parameter space $\mathfrak{R}^D$. The stages of the differential evolution algorithm are shown in Figure 22.

3.3.1 *Initialization of vectors*

DE initializes NP D-dimensional vectors of real-valued parameters. Each vector forms a candidate solution to the optimization problem. We denote the i^{th} vector of the population at the current generation G:

$$\vec{X}_{i,G} = [x_{1,i,G}, x_{2,i,G}, \ldots, x_{D,i,G}] \tag{8}$$

FEATURE	GA	ES	EP
Representation	Binary	Real	Real
Fitness function	Normalized value of the objective function	Value of the objective function	Normalized value of the objective function
Self-adaptation	None	Standard deviation and rotation angles	None
Recombination	One-point crossover, two-point crossover and uniform crossover	Discrete and intermediate	None
Mutation	Secondary operator	Main operator	Only operator
Selection	Probabilistic and based on preservation	Deterministic extinctive or based on preservation	Probabilistic extinctive
Typical Problems	Mixed or continuous optimization	Continuous optimization	Continuous optimization

Tabla 5: Main features of the three main evolutionary computation paradigms.

For each parameter the upper and lower bounds must be specified. The lower and upper bounds are given for $\vec{X}_{min} = x_{1,min}, x_{2,min}, \ldots, x_{D,min}$ and $\vec{X}_{max} = x_{1,max}, x_{2,max}, \ldots, x_{D,max}$, respectively. At generation $G = 0$ we may initialize the j^{th} component of the i^{th} vector as

$$x_{j,i,0} = x_{j,min} + rand_{i,j}[0,1] \cdot (x_{j,max} - x_{j,min}) \tag{9}$$

where $rand_{i,j}[0,1]$ is a random value generated with an uniform distribution between 0 and 1. $x_{j,i,0}$ is instantiated independently for each component of the i^{th} vector.

3.3.2 *Mutation*

The mutant vector, $\vec{V}_{i,G}$, is obtained with the following expression:

$$\vec{V}_{i,G} = \vec{X}_{r_1^i,G} + F \cdot (\vec{X}_{r_2^i,G} - \vec{X}_{r_3^i,G}) \tag{10}$$

where $\vec{X}_{r_1^i,G}$, $\vec{X}_{r_2^i,G}$ and $\vec{X}_{r_3^i,G}$ are vectors chosen randomly from the current population. The indices r_1^i, r_2^i and r_3^i are integers (mutually exclusive) randomly chosen from the range $[1, NP]$. These values must be different from the base vector index i. These index are randomly generated once for each mutant vector. The scaled factor, F, is a positive real number that controls the rate at which the population evolves. F typically lies in the interval $[0.4, 1]$ [17].

3.3.3 *Crossover*

With the crossover operator, the donor vector $\vec{V}_{i,G}$ exchanges its components with the target vector $\vec{X}_{i,G}$ to form the trial vector $\vec{U}_{i,G} = [u_{1,i,G}, u_{2,i,G}, \ldots, u_{D,i,G}]$. DE normally adopts binomial crossover in which the number of parameters inherited from the donor has a (nearly) binomial distribution. Its description is as follows:

$$u_{j,i,g} = \begin{cases} v_{j,i,g} \text{ if}(rand_{i,j}[0,1] \leqslant Cr \text{ or } j = j_{rand}) \\ x_{j,i,g} \text{ otherwise.} \end{cases} \tag{11}$$

The crossover probability, $Cr \epsilon [0, 1]$, is a user defined value that controls the number of parameters inherited from the donor vector.

3.3.4 *Selection*

The next step is to determine whether the target or the trial vector survives to the next generation (at G = G + 1). The selection operation is described as

$$\vec{X}_{i,G+1} = \begin{cases} \vec{U}_{i,G}, \text{ if } f(\vec{U}_{i,G}) \leqslant f(\vec{X}_{i,G}) \\ \vec{X}_{i,G}, \text{ otherwise} \end{cases} \tag{12}$$

where $f(\vec{X})$ is the objective function to be minimized.

3.3.5 *DE Family of Storn and Price*

Actually, the mutation process defines the DE scheme. Let's see how the different DE schemes are named. The general convention used is $DE/x/y/z$,

where DE means "differential evolution", x represents a string denoting the base vector to be perturbed, y is the number of difference vectors considered for perturbation of x, and z denotes the type of crossover (for example, exp: exponential and bin:binomial).

The main mutation schemes are the following (see [17]):

1. "DE/best/1:"

$$\vec{V}_{i,G} = \vec{X}_{best,G} + F \cdot (\vec{X}_{r_1^i,G} - \vec{X}_{r_2^i,G}) \tag{13}$$

2. "DE/target-to-best/1:"

$$\vec{V}_{i,G} = \vec{X}_{i,G} + F \cdot (\vec{X}_{best,G} - \vec{X}_{i,G}) + F \cdot (\vec{X}_{r_1^i,G} - \vec{X}_{r_2^i,G}) \tag{14}$$

3. "DE/best/2:"

$$\vec{V}_{i,G} = \vec{X}_{best,G} + F \cdot (\vec{X}_{r_1^i,G} - \vec{X}_{r_2^i,G}) + F \cdot (\vec{X}_{r_3^i,G} - \vec{X}_{r_4^i,G}) \tag{15}$$

4. "DE/rand/2:"

$$\vec{V}_{i,G} = \vec{X}_{r_1^i,G} + F \cdot (\vec{X}_{r_2^i,G} - \vec{X}_{r_3^i,G}) + F \cdot (\vec{X}_{r_4^i,G} - \vec{X}_{r_5^i,G}) \tag{16}$$

Here r_1^i, r_2^i, r_3^i, r_4^i, and r_5^i are mutually exclusive randomly chosen integers in the range [1, NP], and all are different from the base index i. These indices are randomly generated once for each donor vector. The scaling factor F is a positive control parameter for scaling the difference vectors. $\vec{X}_{best,G}$ is the best individual vector with the best fitness in the population at generation G.

3.4 PREVIOUS RELATED WORK

In general, the previous related work that we reviewed and the proposals of this thesis, use the procedure described next to carry out the fine tuning of weights of the evaluation function. The procedure must create a number of virtual players (usually between 10 and 20), where each virtual player represents a chess program and contains specific (different) weights of its evaluation function. The main objective is to ascertain fitness values of virtual players in order to promote them into successive generations.

The competitive process can be based on one of the following approaches: supervised adjustment, unsupervised adjustment or hybrid adjustment. The

first case, can use games from chess grandmasters, typical chess problems or another chess engine, in order to decide the fitness of the virtual players (for example, fitness can be associated to chess problems that are successfully solved). The second case, uses tournaments among virtual players and those with most wins will be the fittest (this approach is also known as *co-evolution*). Finally, the third approach is a combination of the first two. Based on this classification, we will describe next the previous work related to the contents of this thesis. Sections 3.4.1 and 3.4.2 illustrate the main work related to unsupervised adjustment and supervised adjustment, respectively. Finally, Section 3.4.3 reviews works based on the hybrid approach.

3.4.1 *Works related to unsupervised adjustment*

The works that are most closely related to our own which have been developed using a scheme of unsupervised adjustment, are the following.

Learning To Play the Game of Chess

Thrun [78] developed the program *NeuroChess* which learned to play chess from final outcomes of games with evaluation functions represented by neural networks. This work also included both temporal difference learning [75] and explanation-based learning [25]. *NeuroChess* successfully defeated the *GNU-Chess* program in several hundreds of games. However, the level of play was still poor compared with *GNU-Chess* and human chess players.

A self-learning evolutionary chess program

Fogel et al. [30] used unsupervised adjustment to improve the rating of their baseline chess program in 400 points. This approach used an evolutionary algorithm (evolutionary programing, see Section 3.2.4) within a computer program that learned to play chess by playing games against itself. The authors adjusted the following weights: the material values of the pieces, the piece-square values, and the weights of three neural networks. The method worked as follows.

The procedure started with a population of 20 virtual players (ten parents and ten offspring in subsequent generations). The competitive process was carried out by making each player to play ten games (five as white and five as black) against randomly selected opponents from the population (excluding itself).

The selection mechanism of the evolutionary algorithm used the outcome of the games to decide which virtual players would pass to the following generation. After all 20 players completed their games, the ten best virtual players were retained to become parents of the next generation.

One offspring was created from each surviving parent by mutating all the parental material, the piece-square values, as well as the weights and biases of the three neural networks. They used a Gaussian random variable to mutate the weights of the evaluation function of their chess engine.

The authors used three neural networks focused on the first two rows, the back two rows, and the center of the chessboard (squares c3, d3, e3, f3, c4, d4, e4, f4, c5, d5, e5, f5, c6, d6, e6, and f6), respectively. The idea of these neural networks was to control the player's own territory, the opponent's territory, and the center of the chess board. These neural networks were fully connected feedforward networks with 16 inputs, ten hidden nodes, and a single output node. Both the hidden nodes and the output nodes used the standard sigmoid function (see Section 3.1.3).

They carried out ten independent trials among the best-evolved and the non-evolved players. Fifty generations were executed in each of the ten trials, whereupon the best-evolved player was tested as black against the initial non-evolved chess engine in 200 games.

They found that all trials favored the evolved player over the non-evolved player. For example, in the best trial, the evolved player won 73.7% of the games). This evolved player was named *Blondie25*. *Blondie25* was assessed at six-ply by playing 120 games (60 as black and 60 as white) against the commercial chess program *Chessmaster 8000*. In this case, their program yielded 2437 rating points. Similarly, a series of 12 games (six with white and six with black pieces) was played between the best virtual player and the chess program *Pocket Fritz 2.0* (which plays with the rating of a high-level master). In this case the best evolved player obtained 2550 rating points with nine wins, two losses, and one draw.

Further evolution of a self-learning chess program

Fogel et al. [31] carried out a further evolution of the best-evolved player obtained in their previous work. In this case, they evolved their program during 7462 generations instead of 50. In this case, a 16-game series between this virtual player and *Pocket Fritz 2.0* resulted in 13 wins, 0 losses, and 3 draws, yielding a performance rating of approximately 2650 points.

The definition of the selection mechanism, the competitive process, and the three neural networks are the same as defined in their first work.

Both in the first and in the second experiment, games were played using the alpha-beta algorithm.

They employed the quiescence algorithm to extend the search depth in particular situations. For example, exchange of material, checks to the king, and passed pawns that had reached at least the sixth rank on the board (anticipating pawn promotion). Games were executed until one of the virtual players received checkmate or a draw condition arose. Depending on the outcome of the game, a virtual player obtained one point, half a point or zero points for a win, tie or loss, respectively. Draw conditions were given by the rule of 50 moves (after a pawn's move there are 50 moves to pose a checkmate to the opponent), by the third repetition of the same position and by the lack of victory conditions (e.g., in the fight of a king and a bishop against a king).

Blondie25 competes against Fritz 8.0 and a human chess master

In the two previous papers, the authors used only three minutes on each move for their baseline chess program. However, in a third work [32] they developed a heuristic to manage time by trial and error. With this heuristic, they carried out 24 games between *Blondie25* and *Fritz 8.0*, and their program obtained 2635.33 rating points. Remarkably, their best heuristic developed was different for the black side than for the white one. It is noteworthy that *Fritz 8.0* was ranked at 2752 rating points and was rated number five in the world. Also, they tested *Blondie25* against a human chess master, rated at 2301. In four games, *Blondie25* won three and lost one.

The remainder features of *Blondie25* such as its selection mechanism, evaluation function, and neural networks were equal as in the two other papers.

Differential evolution for tuning a chess evaluation function

Bošković et al. [6] used unsupervised adjustment with co-evolution based on the final outcomes of games to adjust the weights of the evaluation function of their chess program. They used a differential evolution algorithm [70] based on the strategy "DE/rand/2/bin".

They tuned the chess material values and the mobility factor of their evaluation function. The "theoretical" value of the mobility factor is 10 and the "theoretical" values of the pieces are: 300, 330, 500 and 900 for the knight,

bishop, rook and queen, respectively [73]. After 50 generations, the weights matched the values known from chess theory.

An Adaptive Differential Evolution Algorithm with Opposition-Based Mechanisms, Applied to the Tuning of a Chess Program

In their second related work, Bošković et al. [7] used a differential evolution algorithm with unsupervised adjustment to improve the rating of their chess program *BBChess*. Again, they used co-evolution based on the final outcomes of games with adaptation and opposition-based optimization mechanisms.

They adapted the control parameter F present in the mutation process of the differential evolution strategy $rand/2$. This factor is responsible for the exploration and exploitation balance in the evolutionary process. They considered that the efficiency of the tuning process depends on the distance between the solution and the individuals in the first population. Thus, they generated an opposite population $\vec{U_{0,i}}$ defined by the following equations:

$\vec{U_{0,1}} = \{U_{0,i,1}, U_{0,i,1}, \ldots, U_{0,i,D}\}$

$U_{0,i,j} = X_{j,low} + X_{j,high} - X_{0,ij}$

where:

$i = 1, 2, \ldots, NP$

$j = 1, 2, \ldots, D$

$\vec{U_{0,i}}$ represent the opposition individuals of the corresponding initial individuals $\vec{X_{0,i}}$.

Thus, they accelerated the convergence process by increasing the probability of the first generation containing individuals closer to the optimal solution.

History Mechanism Supported Differential Evolution for Chess Evaluation Function Tuning

Bošković et al. [5] used again a differential evolution algorithm with unsupervised adjustment to improve the rating of *BBChess*. They improved their opposition-based optimization mechanisms with a new history mechanism which uses an auxiliary population containing competent individuals. This mechanism ensures that skilled individuals are retained during the evolutionary process. They found that the history mechanism improved the tuning process by 155.2 rating points, on average. They concluded that playing more games enables better comparison among virtual players, decreases noise, and improves the tuning process.

An Evolutionary Approach for the Tuning of a Chess Evaluation Function using Population Dynamics

Kendall and Whitwell [54] used unsupervised adjustment to adjust the weights of the evaluation function of their program. They used an evolutionary algorithm and showed how the outcome of the game (win, loss or draw) could be used to adjust the weights of their chess engine. In their work they derived the theoretical values of the knight, bishop, rook and queen.

Using an Evolutionary Algorithm for the Tuning of a Chess Evaluation Function Based on a Dynamic Boundary Strategy

Nasreddine and Kendall [68] proposed an evolutionary algorithm with an approach called "dynamic boundary strategy" in which the boundaries of the interval of each weight are dynamic (in all previous related works the values of the weights were within a fixed boundary, regardless of how the crossover and mutation operators were applied).

The main idea of maintaining dynamic limits of each weight is to improve the exploration search of the algorithm, and with this, to avoid that the algorithm gets trapped in a local optimum. After 520 generations, the weights obtained with the fittest virtual player were very close to those known in chess theory.

Using genetic programming to evolve chess endgame players

Genetic programming (see Section 3.2.4) has also been used for tuning the weights of a chess evaluation function. Hauptman and Sipper [44] evolved strategies for playing chess end-games. They used unsupervised adjustment with co-evolution based on the outcomes of the games. In their method each individual plays against a fixed number of randomly selected opponents (typically five). The final fitness for each virtual player is the sum of all points earned in the entire tournament for a given generation. Their evolved program could draw against *CRAFTY* which is a state-of-the-art chess engine having a rating of 2614 points.

Evolution of an efficient search algorithm for the mate-in-n problem in chess

In a second work, Hauptman and Sipper [45] evolved entire game-tree search algorithms to solve mate-in-N problems in which the opponent cannot avoid being mated in at most N moves. It is worth noticing that this work does not

adopt the alpha-beta pruning algorithm. Also, they were able to reduce the number of search-tree nodes required to find mates in N moves.

3.4.2 *Works related to supervised adjustment*

Next, we provide a short description of the main works related to supervised adjustment.

Evaluation function tuning via ordinal correlation

Chess Informant[1] publishes the best chess games since 1966 with the advantage that these games are analyzed and commented by their own authors or by chess grandmaster players. Another advantage of this publication is that the comments are given in the standard symbols system shown in Table 6.

Symbol	Meaning
$+-$	White is winning
$\pm$	White has a clear advantage
$\overset{+}{=}$	White has a slight advantage
$=$	The position is equal
$\pm$	Black has a slight advantage
$\overset{=}{+}$	Black has a clear advantage
$-+$	Black is winning

Table 6: Symbols for chess position assessment.

Gomboc and Buro [40] proposed a method for optimizing evaluation functions. Basically, they established a correlation between the human assessment symbols shown in Table 6 and the assessment of the evaluation function. They used 649,698 positions from a Chess Informant to successfully adjust 11 weights of the chess engine Crafty.

Genetic Algorithms for Mentor-Assisted Evaluation Function Optimization

David-Tabibi et al. [19] used a genetic algorithm for automatically tuning the weights of the evaluation function of their chess program. They evolved the

1 http://www.informant1966.com/

organisms of the genetic algorithm to mimic the behavior of another chess program that served as a mentor. With their approach they yield similar performance to that of the expert, with respect to the same set of positions. It is worth mentioning that the rating of the chess mentor was 2700 rating points.

They found that the best organism obtained with this method clearly outperformed its original version by a wide margin. Another important aspect is that they reduced the number of weights in the mentor's evaluation function (while the best organism used over 40, the mentor used over 100).

They used a standard implementation of a genetic algorithm with proportional selection and single point crossover. They handled the following parameters:
Number of generations: 300
Population size: 1000
Crossover rate: 0.75
Mutation rate: 0.002

Expert-Driven Genetic Algorithms for Simulating Evaluation Functions

In [20], David-Tabibi et al. extended their previous approach [19]. In this case, they included an extended set of experiments to assess more accurately the performance of the evolved program. Specifically, they carried out a series of matches between the mentor and the evolved organism. Also, they compared the evolved program against three top commercial chess programs. Also, they talk about their participation in the 2008 World Computer Chess Championship in which they obtained the sixth place.

3.4.3 *Works related to hybrid adjustment*

Only one work has been published using a hybrid adjustment scheme. Its description is the following.

Simulating Human Grandmasters: Evolution and Co-evolution of Evaluation Functions

In this work, David-Tabibi et al. [21] used a genetic algorithm with supervised and unsupervised adjustment to fine-tune the weights of the evaluation function of the chess engine Crafty which is a state-of-the-art chess program. In the supervised adjustment phase the organisms are evolved to mimic the

moves made in a database of chess grandmasters games. In the unsupervised adjustment phase, these evolved organisms are further improved by means of co-evolution.

Basically, they executed the supervised adjustment phase ten times to get the top ten organisms. In this phase they calculated the fitness function for each organism with the following steps:

1. Select a set of positions from chess grandmasters games.
2. For each position each organism calculates its next move. The search is performed at a depth of $1 - ply$.
3. Compare the move suggested by the organism with the actual move played by the chess grandmaster. The fitness of the organism will be the total number of moves matches between the organism and the human grandmaster.

The best organisms serve as the initial population in the unsupervised phase. In the co-evolution step each organism plays four games against each other organism. Their genetic algorithm is run for 50 generations. They used elitism, uniform crossover with a rate of 0.75 and a mutation rate of 0.005.

With this approach, the best organism (after the supervised and unsupervised process) won 60% of the games against the chess program Crafty. It is worth mentioning that the evolved organism consisted of 38 weights, while the chess program Crafty, consisted of over 100 weights.

3.5 FINAL REMARKS OF THIS CHAPTER

In this chapter, we presented a short review of artificial neural networks and evolutionary algorithms. We have also outlined the main works that make use of supervised adjustment, unsupervised adjustment or a combination of both (hybrid approaches), in order to perform the weights adjustment of the evaluation function of a chess engine. These concepts are necessary to present our proposals in Chapters 4, 5 and 6. In Chapter 4, we show an original neural network architecture to obtain the chess positional values. In Chapter 5, we propose an evolutionary algorithm to adjust a small number of weights in our chess engine. Finally, in Chapter 6, we extend the work of Chapter 5 to a larger number of weights and we add a local search engine through the Hooke-Jeeves method to achieve a fine adjustment on the chess pieces values.

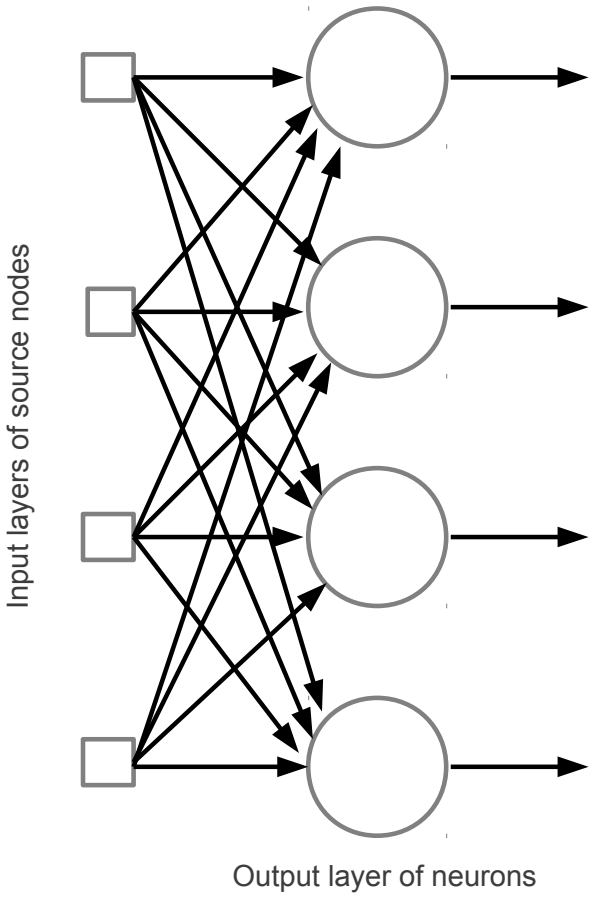

Figure 18: Feedforward network with a single layer of neurons.

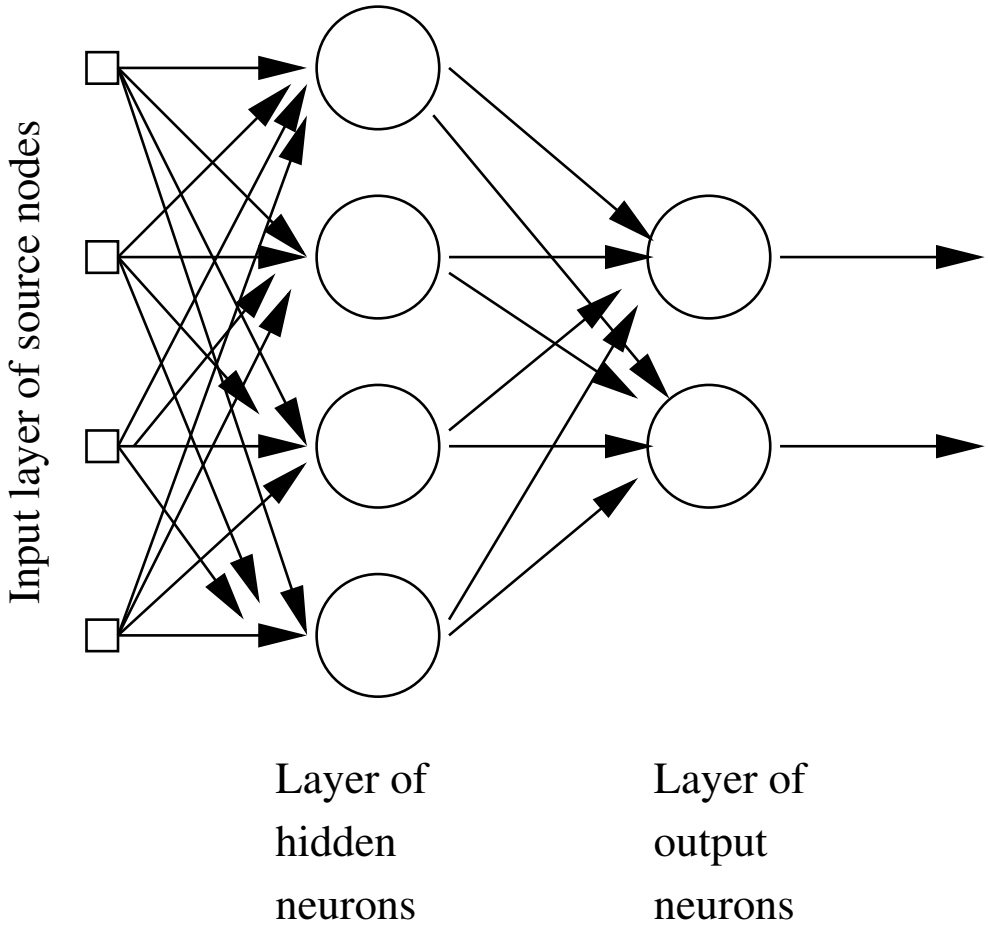

Figure 19: Network with one hidden layer and one output layer. This network is fully connected.

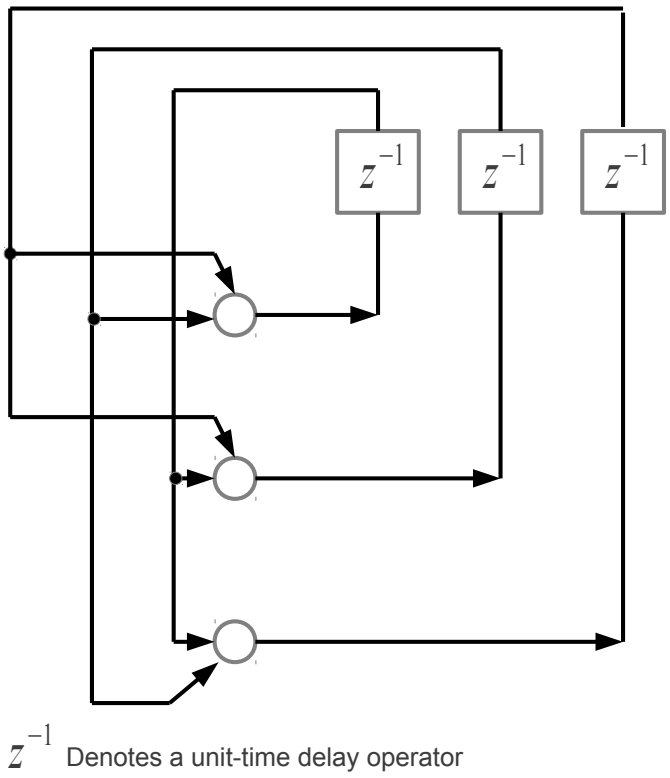

Figure 20: Recurrent network.

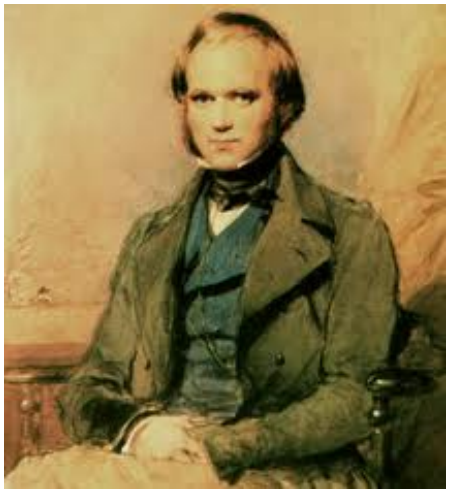

Figure 21: Charles Darwin.

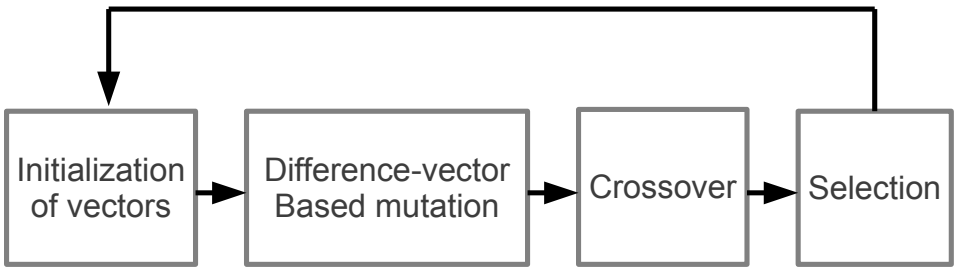

Figure 22: Main stages of the differential evolution algorithm.

"Tactics flow from a superior position."
Bobby Fischer

4

TUNING WEIGHTS THROUGH A NEURAL NETWORK ARCHITECTURE

4.1 INTRODUCTION

This thesis deals with the use of artificial intelligence techniques to carry out the development of a chess engine. In particular, we were interested in the use of neural networks and/or evolutionary algorithms to adjust the weights of the evaluation function of a chess engine.

The mathematical expression of the evaluation function of our chess engine is composed of the sum of material and positional values of the chess pieces. In this work, we proposed an original neural networks architecture, which was used to obtain the positional values of the chess pieces.

In Section 4.2 we will describe the evaluation function that we adopted as well as the concepts of material and positional value of a chess piece. In Section 4.3 we will show our methodology; in particular: our neural network architecture, the components of our evolutionary algorithm and its description. In Section 4.4, we will describe the experiments that we conducted. Finally, in Section 4.5 we will show our final results.

4.2 EVALUATION FUNCTION

As we mentioned in Chapter 2, the evaluation function is one of the most important components of a chess engine. The **evaluation function** that we

used to determine (in a heuristic way) the relative value of a position with respect to one side (white or black pieces) is given by the following expression:

$$f = \sum_{i=1}^{r} m_i + \sum_{i=1}^{q} c_i \times p_i \tag{17}$$

where:
r is the number of pieces in the side under evaluation without considering the king.
q is the number of pieces in the side under evaluation.
m_i is the material value of the piece i.
c_i is the adjustment of the positional value p_i ($c_i = 0.5 \times m_i$).
p_i is the positional value of the piece i. $p_i \in [0, 1]$ (0 represents the worst weights adjustment and 1 represents the best weights adjustment).

4.2.1 *Material values of the chess pieces*

The **material value** of a piece is static and has the values 1, 3, 3, 5 and 9 for the pawn, knight, bishop, rook and queen, respectively [73]. These values have been deduced by chess masters in hundreds of years of practice of this game. They assigned to the pawn the unit value and their experience indicated that the value of the knight, bishop, rook and queen is equivalent to three, three, five and nine pawns, respectively. In our case, we used these values scaled by 100, i.e. 100, 300, 300, 500 and 900 for the pawn, knight, bishop, rook and queen, respectively. It is noteworthy that these values are known as the "theoretical" values of the chess pieces.

4.2.2 *Positional values of the chess pieces*

The **positional value** of a piece is a dynamic value and depends on the characteristics of the position such as mobility, board location, strength, etc. In other works (for example, Fogel et al. [30]) the pieces' positional values are represented by *positional value tables*. The disadvantage of this method is that the positional value of a piece is a static value. For example, in Figure 23 the knight on d2 always has the same value regardless of its location and relationship with the other pieces. In this sense, one of the main ideas of this proposal is that the chess positional values depend directly on the characteristics of the

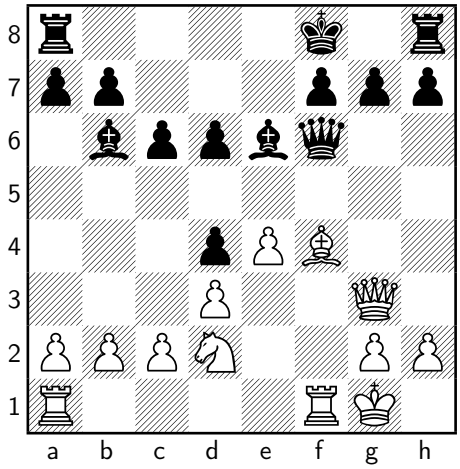

Figure 23: Position to illustrate feature extraction.

position. Of course, as more features are taken into account in calculating the positional value of a piece, the positional value will be more accurate, and therefore, the position will be better evaluated.

4.3 METHODOLOGY

The neural network architecture proposed in this chapter is shown in Figure 24. Next, we will provide its description.

4.3.1 *Neural network architecture*

Our architecture is composed of six neural networks that we use to calculate the positional values of the chess pieces of our chess engine, as defined in equation (17). Each neural network was fully connected and consisted of four nodes in the input layer, nine nodes in the hidden layer and one node in the output layer. The fact that all neural networks have four nodes in the input layer is a mere coincidence, and this number can vary depending on the characteristics chosen to obtain the positional value of a chess piece. The decision to use three layers was based on the demonstration of Hecht-Nielsen [48], which established that any function can be approximated by a three-layer neural network. The decision to use nine nodes in the hidden layer was based on

Kolgomorov's theorem [43], [77] which established that the number of nodes in the hidden layer should be at least $(2i + 1)$, where i denotes the number of nodes in the input layer. As part of our future work, we intend to used an evolutionary algorithm to find the optimal number of hidden units (see [55] and [23]). The hidden nodes used a sigmoid defined by the logistic function $f(y_j) = 1/(1 + exp(-y_j))$, where y_j was the product of the incoming features from the chessboard and the associated weights between the input and hidden nodes, offset by each hidden node's bias term, i.e. $y_j = \sum_{i=1}^{4} f_i \times W_{ij} + \theta_j$, where f_i is the incoming feature i, W_{ij} is the weight between the node i and node j, and θ_j is the bias term of the node j. The value of the output node gives the positional value of a chess pieces within the interval $[0, 1]$. This node also used the logistic function explained before.

Next, we will describe and illustrate the manner in which we obtained the chess positional values through our neural networks architecture.

4.3.1.1 *King's positional value*

The first neural network in Figure 24 obtains the positional value of the king. This network has four input signals that correspond to the following features.

- *Attacking material*. In chess, it is very important to determine the number and type of pieces that are attacking the opposite king. With this feature we tried to represent the attacking material aspect. This refers to the material value of the pieces that are attacking the opposite king. By this, we mean those pieces whose moves act on their opposite king's square or on their opposite king's adjacent squares. For example, in Figure 23, only the queen on f6 attacks the king on g1; therefore, the attacking material corresponding to the white king is 900. Similarly, only the queen on g3 attacks the king on f8; therefore, the attacking material corresponding to the black king is 900.

  In this case, the moves of the pieces can jump to other pieces because this allows to find indirect attacks. This makes possible to detect tactical themes in chess, such as discovered attacks, discovered checks, X-ray attack, among others [71]. For example, in Figure 25, the white side can win the black rook when the bishop on d5 moves to the f3 square.

- *Defending material*. Similarly, in chess it is also important to determine the number and type of pieces that defend their king. With this feature we attempted to represent this aspect. It refers to the material value of the

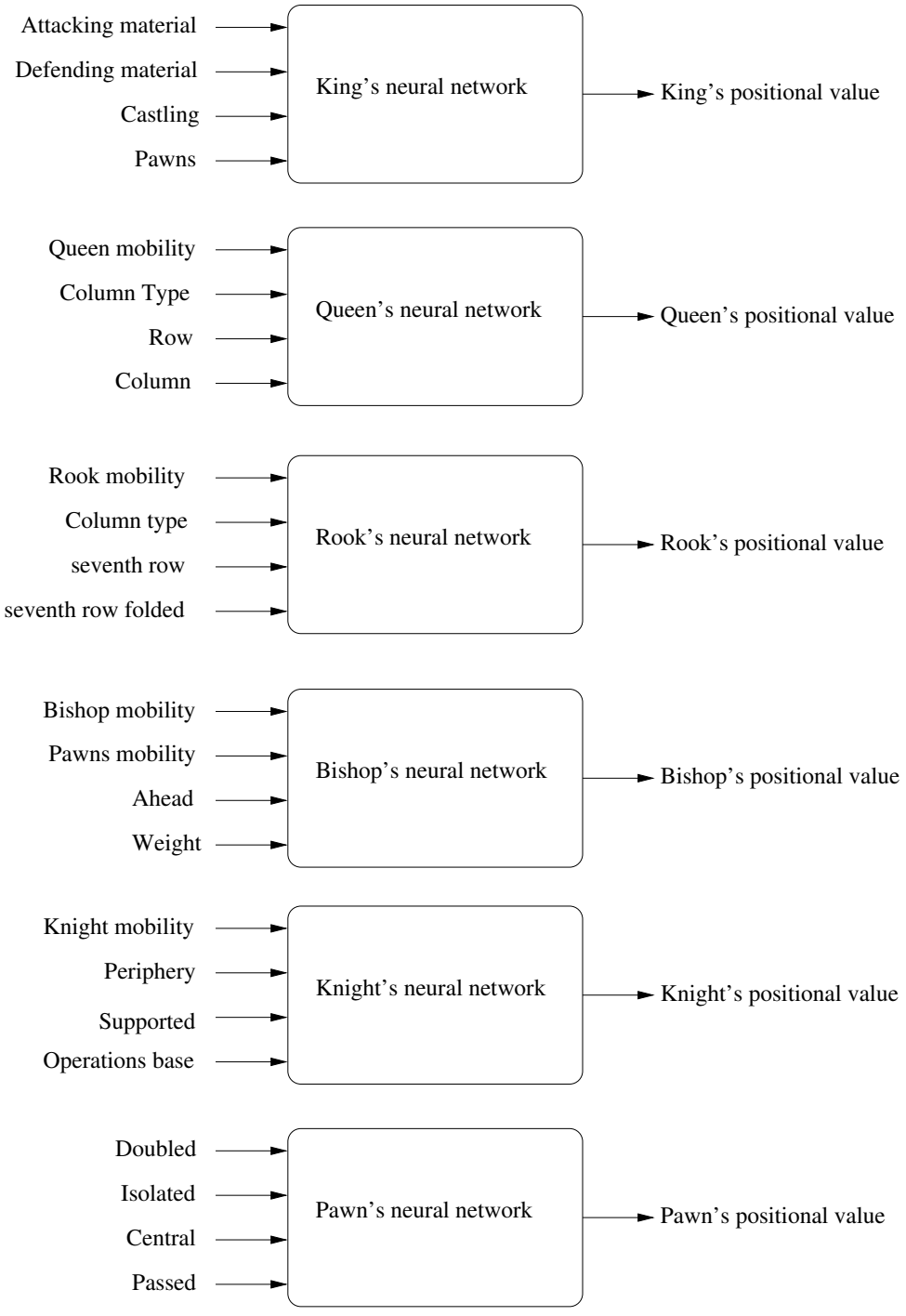

Figure 24: Neural networks architecture used in the evaluation of the pieces' positional values.

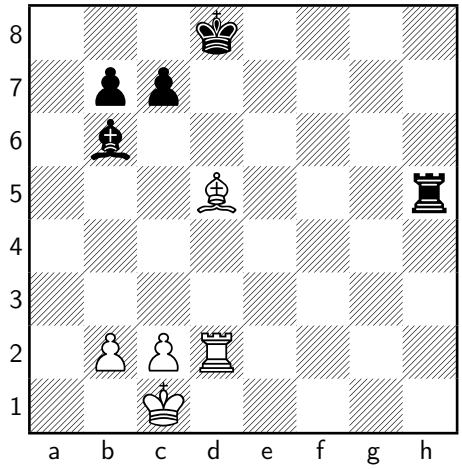

Figure 25: Position to illustrate feature extraction.

pieces that are defending their king. By this, we mean those pieces whose moves act on their king's square or on their king's adjacent squares. For example, in Figure 23, the queen on g3, the rook on a1, the rook on f1, the bishop on f4 and the knight on d2 all defend the white king; therefore, the defending material corresponding to the white king is 2500 ($900 + 500 + 500 + 300 + 300$).

Again, we considered that the moves of the pieces can jump other pieces because our chess engine can solve tactical themes more easily. For example, in Figure 26, the white side prevents checkmate because the white queen defends the f1 square.

- *Castling*. It is a binary value. It is one if and only if the king is castled. In Figure 26, this value is one for the white king.
- *Pawns*. It is the number of pawns located on its king's adjacent squares. In Figure 26, this value is two for the white king.

4.3.1.2 *Queen's positional value*

The second neural network in Figure 24 obtains the positional value of the queen. This network has four input signals that correspond to the following features.

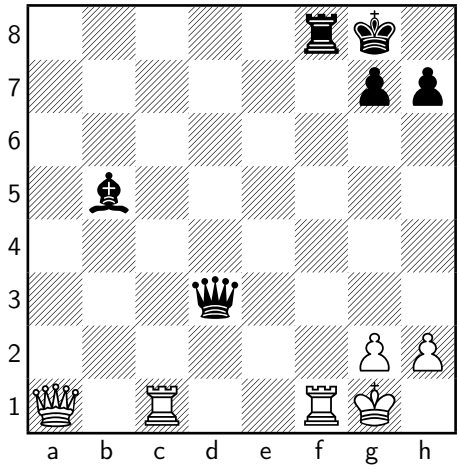

Figure 26: The white queen prevents checkmate on f1 square.

- *Queen mobility*. It is the number of moves of the queen. If more moves a chess piece has, greater will be its associated value. In Figure 27, this value is 16 for the white queen and is four for the black queen; therefore, the white queen is much stronger than the black queen.

- *Column type*. It is zero if on the queen's column there are no pawns, it is one if on the queen's column there are adversary pawns and the queen is in front of its pawns (if any), and it is two if on the queen's column there are pawns of both sides and the queen is behind any of its pawns.

  It is expected that the value of a queen in a column without pawns (or on front of its pawns) is greater than the value of a queen behind its pawns. For example, in Figure 27, the value of the queen on g3 is greater than the value of the queen on a8. In Figure 27, this value is 1 for the white queen.

- *Row*. It refers to the row occupied by the queen. In Figure 27, this value is three for the white queen.

- *Column*. It refers to the column occupied by the queen. In Figure 27, this value is seven for the white queen.

  The knowledge of the queen's row and column is very important because a queen on the center of the board will be stronger than a queen on the

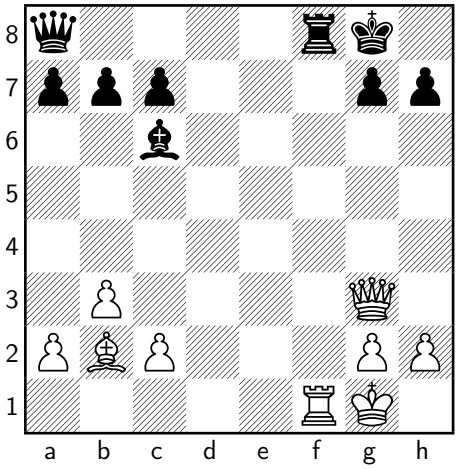

Figure 27: The value of the queen of g3 is greater than the value of the queen on a8.

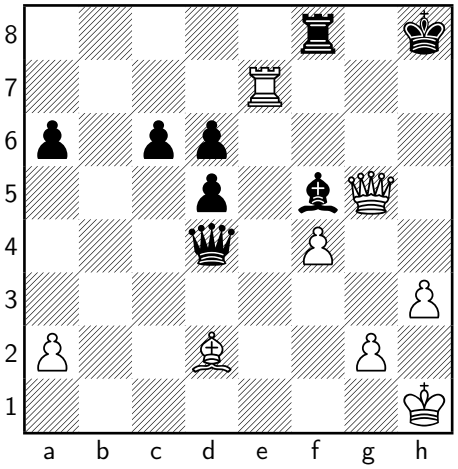

Figure 28: The white rook on the seventh row permits to the white side win the black queen.

periphery of the board. Again, in Figure 27 the value of the queen on g3 is greater than the value of the queen on a8.

4.3.1.3 *Rook's positional value*

The third neural network in Figure 24 obtains the positional value of the rook. This network has four input signals that correspond to the following features.

- *Rook mobility*. It is the number of moves of the rook. By the same arguments given for the queen, it is important to know the mobility of the rook. In Figure 27, this value is 12 for the rook on f1.
- *Column type*. See the definition of the column type for the queen. In Figure 27, this value is 0 for the rooks on f1 and f8.
- *Seventh row*. It is a binary value. It is one if and only if the rook is on the seventh row. A rook on the seventh row is usually very strong because it allows the gain of material. For example, in Figure 28, the white rook on the seventh row allows the white side to win the black queen through the sequence of moves: *1. Qh4+ Kg8 2. Qg3+ Kh8 3. Bc3*. This feature is equal to one for the white rook in this figure.

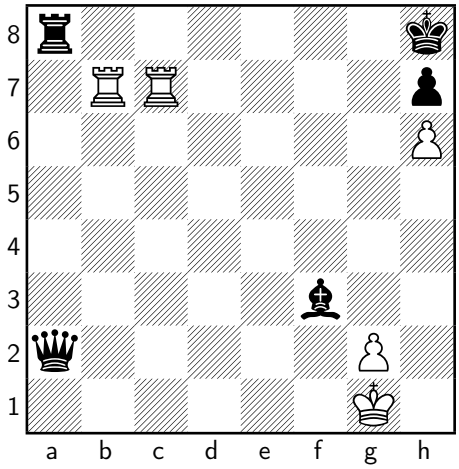

Figure 29: The black king receives checkmate by the white rooks on b7 and c7.

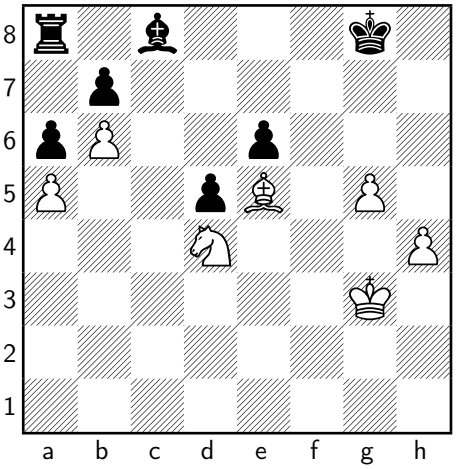

Figure 30: The value of the bishop on e5 is greater than the value of the bishop on c8.

- *Seventh row folded*. It is a binary value. It is one if and only if there are at least two rooks on the seventh row. Rooks folded on the seventh row allow combinations that lead to the gain of material or checkmate to the opposite king. For example, in Figure 29, the black side receives checkmate after the moves *1. Rxh7+ Kg8 2. Rbg7+ Kf8 3. Rh8+ Qg8 4. Rhg8++* . This value is one for the rook on b7 and for the rook on c7 in this figure.

4.3.1.4 *Bishop's positional value*

The fourth neural network in Figure 24 obtains the positional value of the bishop. This network has four input signals that correspond to the following features.

- *Bishop mobility*. It is the number of moves of the bishop. By the same arguments given for the queen, it is important to know the mobility of the bishop. In Figure 28, this value is six for the bishop on d2.
- *Pawn's mobility*. The pawn's mobility of a bishop is the number of moves of its pawns which obstruct (or may obstruct) its mobility. For example, in Figure 30 the number of the moves is zero and two for the bishop on c8 and e5, respectively.

- *Ahead*. It is the number of pawns which are in front of their bishop and obstructing its movement. These pawns are those located in a row number higher than the row number occupied by the bishop for the white pieces. Similarly, these pawns are those located in a row number lower than the row number of the bishop for the black pieces. In Figure 30, this value is four for the black bishop because the pawns on a6, b7, e6 and d5 obstruct its movement. Similarly, this value is one for the white bishop because the pawn on b6 obstructs its movement.

- *Weight*. Any chess expert will notice in Figure 23 that the pawn on d4 obstructs more the movement of the bishop on b6 than the pawn on a7. Each square on the board is assigned a numeric value that reflects the degree of obstruction of a pawn on the bishop's movement. Table 7 shows the initial values of the weights of black pawns than obstruct the black bishop's movement (these values were taken from [42]), and Table 8 shows the initial values of the weights of the white pawns that obstruct the black bishop's movement (these values have been assigned by an expert in chess). The weights of the white pawns and the black pawns that obstruct the white bishop's movement are the mirror of Tables 7 and 8, respectively. The weights in Tables 7 and 8 are initial values and must be evolved with our method to find their optimal values. It is worth mentioning that it is not necessary to evolve the weights on the first and eighth row because there will never be pawns on these squares. In Figure 23, the weight for the bishop on b6 is 26 $(2+4+16+4)$.

4.3.1.5 *Knight's positional value*

The fifth neural network in Figure 24 obtains the positional value of the knight. This network has four input signals that correspond to the following features.

- *Knight mobility*. It is the number of moves of the knight. By the same reasons given for the queen, it is important to know the mobility of the knight. In Figure 31, this value is eight for the knight on d6.

- *Periphery*. It is a binary value. It is one if and only if the knight is on the periphery of the board (first row, eighth row, first column or eight column). In Figure 31 the black night is bad because is on the periphery of the board. In this figure, this value is one for the black knight.

8	0	0	0	0	0	0	0	0
7	2	4	4	8	8	4	4	2
6	2	4	8	16	16	8	4	2
5	2	4	12	24	24	12	4	2
4	2	4	4	4	4	4	4	2
3	2	2	2	2	2	2	2	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
	a	b	c	d	e	f	g	h

Tabla 7: Initial weight values of black pawns than obstruct the black bishop's movement.

- *Supported*. It is a binary value. It is one if and only if the knight is supported by one of its pawns. In Figure 31, the knight on d6 is better than the knight on g6. In Figure 31, this value is one for the knight on d6.
- *Operations base*. It is a binary value. It is one if and only if the knight is on an *operations base*. A knight is on an *operations base* if it cannot be evicted from its position by an opponent pawn. This value is one for the knight on d6.

In conclusion, the value of a centralized knight, that has all its moves supported by one of its pawns and that cannot be evicted from its position by an opponent pawn will be greater than another knight that lacks in some of these aspects.

4.3.1.6 *Pawn's positional value*

The sixth neural network in Figure 24 obtains the positional value of the pawn. This network has four input signals that correspond to the following features.

- *Doubled*. It is a binary value. It is one if and only if there are at least two pawns located in the same column. A doubled pawn becomes weaker because it can not be defended by another pawn. In Figure 32, this value is one for the pawn on g7.
- *Isolated*. It is a binary value. It is one if and only if a pawn cannot be defended by another pawn. An isolated pawn becomes weaker because

8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	1	1	1	1	1	1	0
5	0	1	2	2	2	2	1	0
4	0	1	2	2	2	2	1	0
3	0	1	1	1	1	1	1	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
	a	b	c	d	e	f	g	h

Tabla 8: Initial weight values of the white pawns than obstruct the black bishop's movement.

it can be easily attacked. In Figure 32, this value is one for the pawn on g7.

- *Central*. It is a binary value. It is one if and only if the pawn is on any of the following squares: c4, c5, d4, d5, e4, e5, f4 or f5. If a pawn is central, its associated value will be greater. In Figure 32, this value is one for the pawn on d5.

- *Passed*. It is a binary value. It is one if and only if the pawn cannot be stopped by an opponent pawn. Creating passed pawns is very important since this fact limits the defense of the adversary to prevent its possible coronation. In Figure 32, this value is one for the pawn on d5.

The ideal case of a pawn is when it is in the center of the chessboard, is passed, is not doubled and is not isolated.

It is worth noticing that the values obtained with our neural network architecture are conceived to correspond to the chess pieces' positional values of a mid-game.

4.3.2 *Components of our evolutionary algorithm*

The components of our evolutionary algorithm are the following.

- **Evaluation function**. The evaluation function of our evolutionary algorithm was already explained in Section 4.2.

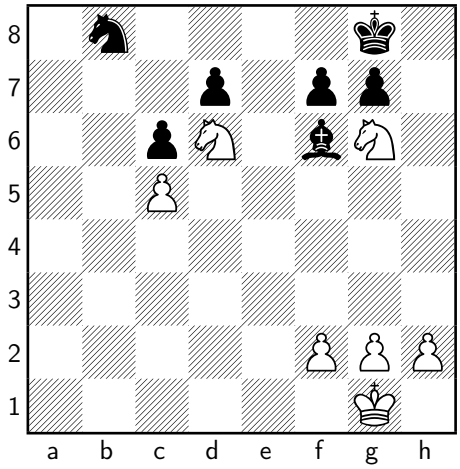

Figure 31: The value of the knight on d6 is greater than the value of the knight on g6, and this is greater than the value of the knight on b8.

- **Representation**. In our evolutionary algorithm we used real-numbers encoding, where a chromosome is composed by the weights of the neural network architecture shown in Figure 24 and the weights of the pawns which obstruct the bishop's mobility. We used 426 weights whose description is given below.
  - **Weights of the neural networks**. Each neural network was fully connected and has four input nodes, nine hidden nodes and one output node. In this way, each neural network has $4 \times 9 + 9 \times 1$ weights plus $9 + 1$ bias. In total, each neural network has 55 weights. Because we used six neural networks, we have $55 \times 6 = 330$ weights.
  - **Weights of the pawns which obstruct the bishop's mobility**. In this case we have Tables 7 and 8, so we have 96 (48×2) weights.
- **Population**. The initial population of our evolutionary algorithm consisted of $n = 20$ (10 parents and 10 offspring in subsequent generations) virtual players whose weights were randomly initialized using an uniform distribution within their allowable bounds.
- **Survivor selection mechanism**. This mechanism is explained in Section 4.3.3.

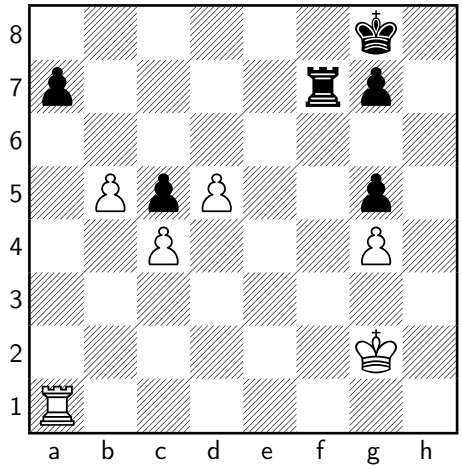

Figure 32: Position to illustrate feature extraction.

- **Operators**. One offspring was created from each surviving parent by mutating all weights and biases by adding a Gaussian random variable with zero mean and a standard deviation of 0.05, as Chellapilla and Fogel did in [13]. If, after mutating a weight, its value falls outside the range, this value is re-set to the nearest extreme of its range. Since we adopted evolutionary programming no crossover operator is employed.

4.3.3 *Our evolutionary algorithm*

Figure 33 outlines the evolutionary algorithm adopted to adjust the neural networks' weights in order to compute the pieces' positional values. The first module, called "initialize population", assigns initial random weights to the neural networks and the weights of the pawns that obstruct the bishop's mobility. The features of the position (inputs of the neural network) are obtained in the module "Features extraction".

The module "Play tournament" coordinates a tournament between n virtual players (in our case $n = 20$). Each virtual player is allowed to play $n/2$ games with randomly chosen opponents. The side (either black or white) is also chosen at random. Games are executed until one of the virtual players receives checkmate or a draw condition arises. Depending on the outcome of the game, a virtual player obtains one point, half a point or zero points for a win, tie or

loss, respectively. Draw conditions are given by the rule of 50 moves (after a pawn's move there are 50 moves to pose a checkmate to the opponent), by the third repetition of the same position or by the lack of victory conditions (e.g., in the fight of a king and a bishop against a king). This module uses the chess engine described in Section 2.7 in the page 26.

After finishing the tournament, the "Selection" module applies the survivor selection mechanism to choose the $n/2$ virtual players having the highest number of points, and in the module "Mutation" these virtual players are mutated to generate the remaining $n/2$ virtual players. Finally, the evolutionary algorithm (based on evolutionary programming [33]), continues running for 50 generations.

The weights and biases of the neural networks were initialized in the range $[-15, 15]$ and the weights of the pawns which obstruct the bishop's mobility were initialized in the range $[0, 20]$ (we carried out different experiments, and we found that the ranges of these weights fall into these intervals).

4.4 EXPERIMENTAL DESIGN

The experiments were carried out on a PC with a 64-bits architecture, having two cores running at 2.8 GHz each and 3 GBytes in RAM. The programs were compiled using *g++* in the OpenSuse 11.1 operating system. For the experiments reported next, we used the opening book *Olympiad.abk* both for the virtual players and for the chess engine Rybka 2.3.2a (see the web page `http://www.rybkachess.com/`).

4.5 EXPERIMENTAL RESULTS

In the following experiments we carried out ten runs for each of the following values of $c = 0.1, 0.2, \ldots, 1.0$ (see equation (17) in the page 64), and we found that for the value of $c = 0.5$ our chess engine obtained the best rating.

4.5.1 *Experiment A*

This experiment consisted of performing ten runs, and in each of them we had 20 virtual players that were evolved during 50 generations. The weights of the virtual players were randomly initialized within the allowable bounds with a different seed for each run. At the end of each run, we carried out 200 games between the best virtual player at generation 50 and the best virtual player at

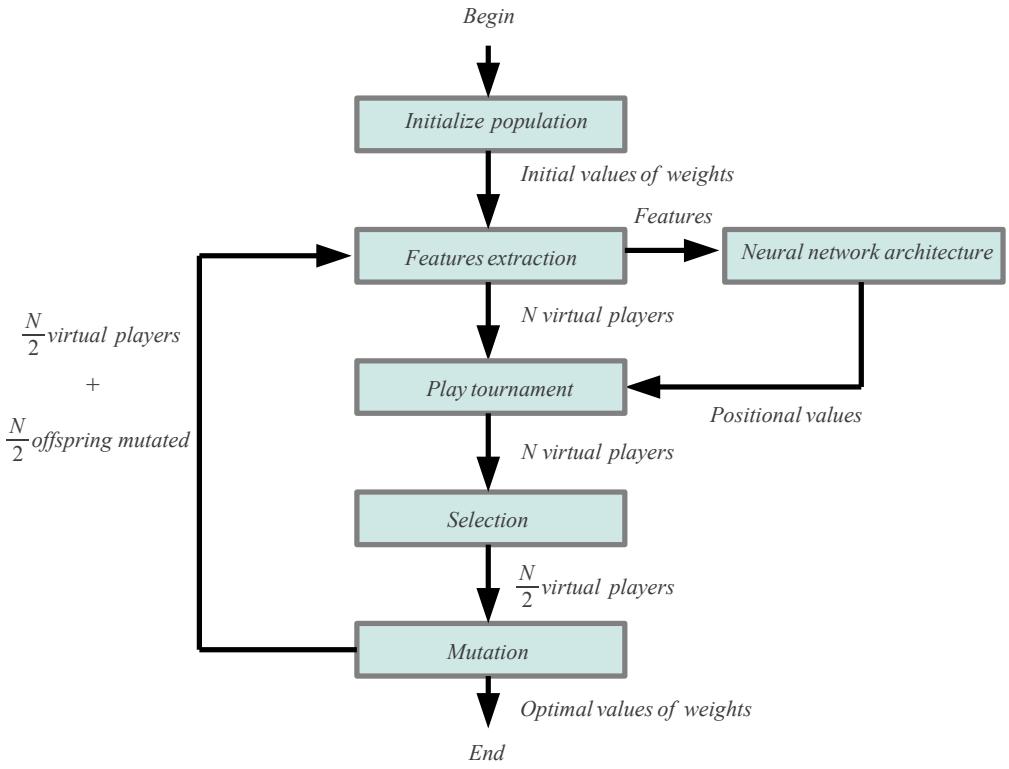

Figure 33: Flowchart of the evolutionary algorithm adopted in this work.

generation 0; Table 9 shows these results. For example, in run 1 the best player at generation 50 won 180, drew 14 and lost 6 games against the best player at generation 0 (the percentage of games won by the best player at generation 50 was 93.50%). The best result corresponds to the third run, in which the best virtual player at generation 50 won 185, drew 12 and lost 3 games against the best player at generation 0 (the percentage of games won by the best player at generation 50 was 95.50%). In this experiment we used a search depth of four plies (1 ply corresponds to the move of one side),

4.5.2 *Experiment B*

In this experiment, the best virtual player at generation 0, was called $\texttt{player}_0$ and played 60 games against the chess engine Rybka 2.3.2a using each of the

Run	Wins	Draws	Losses	Wins%
1	180	14	6	93.50%
2	171	26	3	92.00%
3	185	12	3	95.50%
4	169	28	3	91.50%
5	174	25	1	93.25%
6	176	19	5	92.75%
7	182	16	2	95.00%
8	183	15	2	95.25%
9	178	18	4	93.50%
10	168	28	4	91.00%

Tabla 9: Number of games won, drawn and lost for the best virtual player at generation 50 against the best virtual player at generation 0.

following ratings: 2300, 2100, 1900 and 1700. The histogram of results is shown in Figure 34. For example, $player_0$ won, drew and lost 0, 3 and 57 games, respectively, against Rybka 2.3.2a playing at 2300 rating points; $player_0$ won, drew and lost 4, 6 and 50 games, respectively, against Rybka 2.3.2a playing at 2100 rating points. The same experiment was carried out with the best virtual player for the ten runs in Table 9. This virtual player was called $player_{50}$ and corresponds to the third run in this table. The results against the chess engine Rybka 2.3.2a are shown in Figure 35. In this figure we can see that $player_{50}$ won, drew and lost 14, 10 and 36 games, respectively, against Rybka 2.3.2a playing at 2300 rating points; $player_{50}$ won, drew and lost 26, 22 and 12 games, respectively, against Rybka 2.3.2a playing at 2100 rating points.

Based on these played games, we used the Bayeselo tool[1] to estimate the ratings of our virtual players using a minorization-maximization algorithm [51]. The obtained ratings are shown in Table 10. The columns **Rank**, **Name**, **Elo**, **+**, **-**, **Games**, **Score**, **Opposition** and **Draws** give the classification obtained, the name, the rating obtained, the lower bound of the confidence interval (with 95% confidence), the upper bound of the confidence interval (with 95% confidence), the number of games carried out, the percentage of games won, the average rating of the opponents and the percentage of ties made by each virtual player and the chess engine Rybka2.3.2a. For example, $Rybka_{2300}$ played

1 http://remi.coulom.free.fr/Bayesian-Elo/

120 games, won 83% games, with an average rating of the opponents of 1961 points and tied 11% of the games. Rybka won the first place, with a rating of 2309 points which is estimated between $2309 + 64$ and $2309 - 59$ with 95% confidence. In this table, we can see that the rating for the virtual player $player_0$ was 1745, and the rating for the virtual player $player_{50}$ was 2178, representing an increase of 433 rating points between the non-evolved and the evolved virtual players after 50 generations for the third run of Table 9 (2178 ratings points is a value close to a chessmaster level [30]).

In this experiment we used a search depth of six plies for the chess engine Rybka2.3.2a, as well as for $player_0$ and $player_{50}$.

It is worth noticing that Thrun [78] employed one neural network with 175 input nodes, 165 hidden nodes and 175 output nodes within his program *NeuroChess*. NeuroChess successfully won 11% of the games versus the program *GnuChess* (which has about 2300 rating points), and our chess program won 31.6% of the games versus Rybka 2.3.2a playing at 2300 rating points. In another previous related work, Fogel et al. [30] employed three neural networks, each one having 16 input nodes, 10 hidden nodes and 1 output node. The strength of their program was about 2550 rating points.

Rank	Name	Elo	+	-	Games	Score (%)	Opposition	Draws (%)
1	$Rybka_{2300}$	2309	64	59	120	83%	1961	11%
2	$Player_{50}$	2178	38	37	240	69%	1997	18%
3	$Rybka_{2100}$	2097	51	50	120	63%	1961	23%
4	$Rybka_{1900}$	1883	51	52	120	41%	1961	16%
5	$Player_0$	1745	40	41	240	25%	1997	12%
6	$Rybka_{1700}$	1699	56	60	120	25%	1961	9%

Tabla 10: Ratings on the third run against Rybka2.3.2a.

In the previous experiments each virtual player was allowed to play $n/2$ games with randomly chosen opponents. It is also noteworthy that these experiments were repeated allowing each virtual player to play against the remaining virtual players (in total $n - 1$ games), and in this case, the best virtual player with $n - 1$ games was only three points higher than the best virtual player with $n/2$ games.

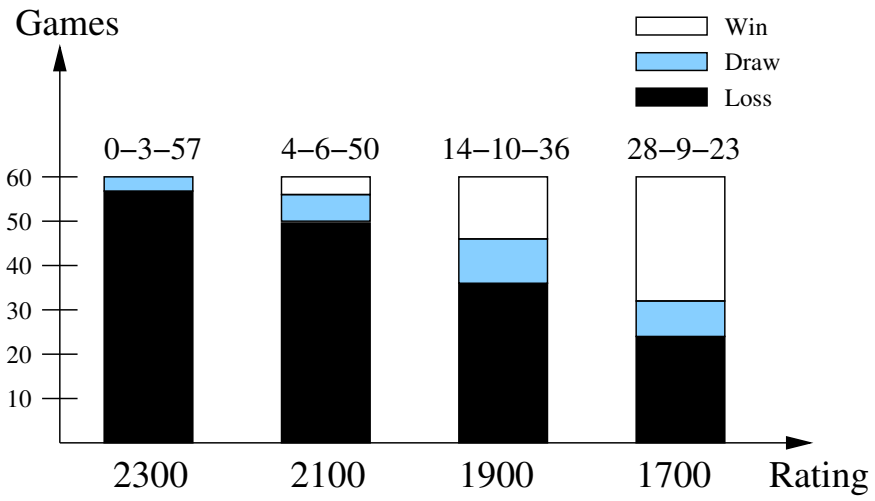

Figure 34: Histogram of wins, draws and losses for the best virtual player at generation 0 (player0) against Rybka 2.3.2a.

Finally, in Table 11 we show the weights of the black pawns than obstruct the black bishop's movement for virtual player $player_{50}$. In this table, we can confirm the chess knowledge which states that a bishop is "bad" if the pawns that obstruct its movement are in the middle of the board [42].

4.5.3 *Discussion of the results*

The idea of experiment A was to carry out games between the evolved virtual player and the non-evolved virtual player with the aim of validating that the first player indeed performed better than the second. The idea of experiment B was to assess the rating of the virtual players adopted as well as that of a commercial chess engine (in this case, we used Rybka). From experiment A, we concluded that the evolved virtual player was indeed better (in terms of performance) than the non-evolved player. In experiment B, we concluded that the rating achieved by the commercial chess engine Rybka, matched its expected value (for example, for Rybka2300, the obtained rating was 2309). Experiments of this sort have also been reported by other authors (see for example, [30], [31], [5] and [21]).

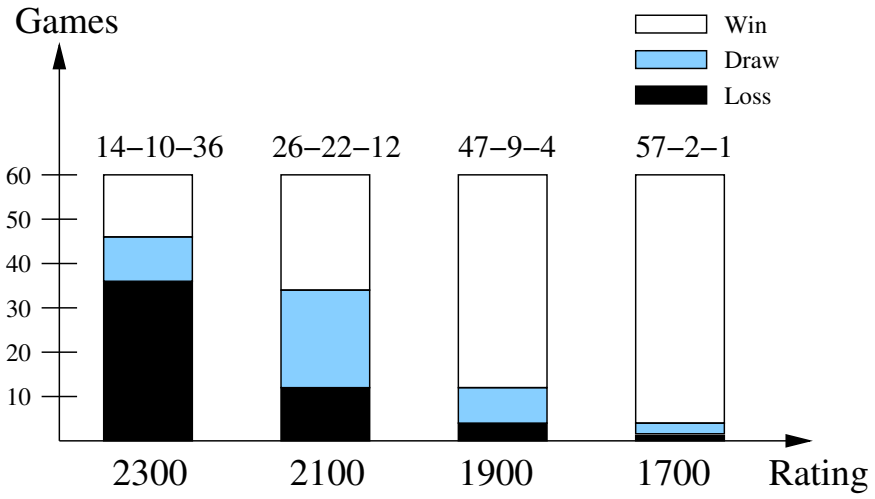

Figure 35: Histogram of wins, draws and losses for the best virtual player at generation 50 ($player_{50}$) against Rybka 2.3.2a.

4.6 FINAL REMARKS OF THIS CHAPTER

In this chapter we proposed a neural network architecture to obtain the positional values of chess pieces. Also, we presented the evolutionary algorithm to tune the weights and biases of these neural networks and the weights which obstruct the bishop's movement.

With this proposal our chess engine reached a rating of 2,178 points, and although this value is really good (it corresponds to the level of an expert in chess by the United States Chess Federation). We are still relatively far from the main objective of this thesis. We can continue to test other alternatives as evolve the parameter c_i or carry out more runs, among others, but in this moment, we prefer to set aside neural networks to test exclusively evolutionary algorithms because they are also belong of the main objectives of this thesis.

8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
7	1.96	3.83	4.10	8.26	8.08	4.01	3.93	1.57
6	2.24	4.09	7.99	16.25	16.19	7.89	3.89	1.76
5	1.97	4.09	12.21	23.99	23.85	12.31	4.31	2.06
4	1.64	3.74	4.36	4.20	4.23	3.79	3.93	1.99
3	0.13	2.14	1.90	2.31	2.21	2.45	2.19	0.15
2	0.01	0.17	0.00	0.00	0.00	0.55	0.10	0.23
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	a	b	c	d	e	f	g	h

Tabla 11: Final weight values of black pawns than obstruct the black bishop's movement.

"Excelling at chess has long been considered a symbol of more general intelligence. That is an incorrect assumption in my view, as pleasant as it might be."

Garry Kasparov

5

TUNING WEIGHTS WITH A DATABASE OF CHESS GRANDMASTER GAMES

5.1 INTRODUCTION

In Chapter 4, we proposed a neural network architecture to carry out the weights adjustment of our chess engine. Using this architecture, our chess engine increased its rating in 433 points (reaching a performance of 2178 points). This value is not bad, but it is the value that we were able to reach with the neural network architecture. For this reason, from this moment on, we proceed to test exclusively evolutionary algorithms.

In fact, in this chapter we propose an evolutionary algorithm that adjust the material values of chess pieces and the mobility factor. In case that we find evidence that the evolutionary algorithm yields good results (that is to say, the material values of chess pieces match their "theoretical" values), in the next chapter we will proceed to adjust a greater number of weights in order to increase the rating of our chess engine.

Most of the previous related work that has been reported in the specialized literature [30, 31, 32, 6, 7, 54, 68] adopted tournaments among virtual players from which the final result of each game (win, loss or draw) is used for deciding which players will pass to the following generation.

In the work reported in this chapter, we carried out the automatic tuning of the weights of our evaluation function using an evolutionary algorithm. The selection mechanism of the proposal presented here uses games from chess

grandmasters to decide which virtual player will pass to the following generation. Our results indicate that the weight values obtained by our proposed approach match the values that are known from chess theory.

The remainder of this chapter is organized as follows. The chess engine adopted for our experiments is described in Section 5.2. The methodology, the evolutionary algorithm and its components are described in Section 5.3. Finally, in Section 5.4, we present our experimental results.

5.2 CHESS ENGINE

The components of our chess program were described in Section 2.7 in the page 26. The only difference is the type of evaluation function employed. Now, we adopt the evaluation function used by Bošković et al. in [6]. This function is:

$$eval = X_m(M_{white} - M_{black}) + \sum_{y=0}^{5} X_y(N_{y,white} - N_{y,black}) \tag{18}$$

In this equation, X_y represents the weights for all pieces except for the king. The king's weight was not taken into account because in eq. (18), its associated term is zero (there is always a king for each side on the board). M_{white} represents the number of available moves (mobility) for the white pieces, and M_{black} represents the mobility for the black pieces. X_m is the mobility weight. $N_{y,white}$ and $N_{y,black}$ are the number of y pieces for the white or the black pieces, respectively. y can denote a queen, rook, bishop or knight. We used the material value of the pawn as a reference by assigning it a value of 100.

The main aim of the algorithm described in this chapter is to show that the weights of the evaluation function can be tuned using an evolutionary algorithm, so that they closely match the values derived from chess theory. This sort of evaluation function is relatively simple, but still provides a reasonably good search strategy for our chess engine. It is worth adding that the training of our search engine was conducted using a database of games from chess grandmasters.

5.3 METHODOLOGY

Our proposed approach is based on an evolutionary algorithm which has a selection mechanism based on a database of chess grandmasters games. The

idea is that the weights adopted in our evaluation function are such that the move performed is equal to the one that was performed by a human chess master in a particular game from the database. This similarity is used to decide which virtual player (individuals in the population) will pass to the following generation.

5.3.1 *Components of our evolutionary algorithm*

The main components of an evolutionary algorithm were described in Section 3.2.2 in page 39. Next, we will give the details of these components for the evolutionary algorithm proposed in this chapter.

- **Representation**. As in Chapter 4, we adopted evolutionary programming [33] to carry out the weights adjustment of our chess engine. Our algorithm used real numbers encoding, where a chromosome is composed by the sequence of weights of equation (18), as shown in Figure 36. In fact, this is the fundamental set of weights that any chess engine should tune. For this reason, we have chosen them to show experimentally, that the evolutionary algorithm proposed in this chapter can adjust the weights of our chess engine.

Figure 36: Chromosome adopted in our evolutionary algorithm.

- **Population**. The population of the algorithm was initialized with N virtual players ($\frac{N}{2}$ parents and $\frac{N}{2}$ offspring in subsequent generations). The weight values for these virtual players were random values generated with an uniform distribution within the allowable bounds. The allowable bound for each piece and for each mobility weight are described in Section 5.4.
- **Survivor selection mechanism**. This mechanism is explained in Section 5.3.2.
- **Operators**. One offspring was created by mutating all weights from each surviving parent with a probability of 90% (we carried out several runs

using mutation rates of 80%, 85%, 90%, 95% and 100%, and found that 90% produced the best convergence and standard deviation values). We mutated the weights shown in Figure 36.

We adopted Michalewicz's non-uniform mutation operator [66]. In this operator, the mutated weight $V_k^{'}$ (obtained from the previous weight V_k) is obtained with the following expression:

$$V_k^{'} = \begin{cases} V_k + \Delta(t, UB - V_k) & \text{if R=TRUE} \\ V_k - \Delta(t, V_k - LB) & \text{if R=FALSE} \end{cases} \tag{19}$$

where the weight V_k is within the range $[LB, UB]$ and $R = flip(0.5)$. The function $flip(p)$ simulates the tossing of a coin and returns TRUE with a probability p. Michalewicz suggests using:

$$\Delta(t, y) = y * (1 - r^{(1-t/T)^b}) \tag{20}$$

where r is a random real number between 0 and 1. T is the maximum number of generations and b is a user-defined parameter. In our case, we used $b = 2$.

If, after mutating a weight, its value falls outside the range, this value is re-set to the nearest extreme of its range. Since we adopted evolutionary programming, no crossover operator is employed in our case.

5.3.2 *Evolutionary algorithm*

Fig. 37 shows the flow chart of our proposed evolutionary algorithm for tuning the weights of the evaluation function given in eq. (18).

First, the weights of N virtual players are initialized with random values within their corresponding boundaries. Subsequently, a virtual player's score is incremented in one for each move of the P games on the database for which the virtual player did the same action as the human chess master.

The value of the parameter P is provided by the user and refers to the number of games that will be (randomly) chosen from the database to calculate the score of a virtual player for a generation. The **survivor selection mechanism**

chooses the N/2 virtual players that achieved the highest score. These virtual players are allowed to pass to the next generation and, consequently, will be allowed to generate offspring using mutation, in order to give rise to the new population of N virtual players. In our experiments, this procedure is repeated by 50 generations.

The procedure for computing the score of each virtual player is described in Algorithm 9. Line 1 gets the set S which consists of P games chosen at random from the database. Parameter P ranges from 1 to the number of games available in the database (in our case, 312). In lines 2 to 4, we establish the score counter to zero for each virtual player. Line 5 chooses d training games from S. Line 6 sets the starting position of the game d. Line 7 chooses the next move m from the game d. Finally, each virtual player calculates his next move n, and if this move matches the move m, this virtual player increases his score in 1.

Algorithm 9 scoreCalculation()

```
1:  S = chooseGames(P)
2:  for each virtual player i do
3:     score[i] = 0
4:  end for
5:  for each game d in S do
6:     setPosition(d)
7:     for each move m in game d do
8:        for each virtual player i do
9:           n = nextMove(i)
10:          if m == n then
11:             score[i]++
12:          end if
13:       end for
14:    end for
15: end for
```

5.3.3 *Database of games*

The database that we adopted consists of 312 games taken from the Linares super tournament in its editions 1999, 2001, 2002, 2003, 2004, 2005, 2008 and 2010. These games can be downloaded from the site *http://www.chessbase.com/*. Clearly, the database can be expanded so that a more robust tuning of weights

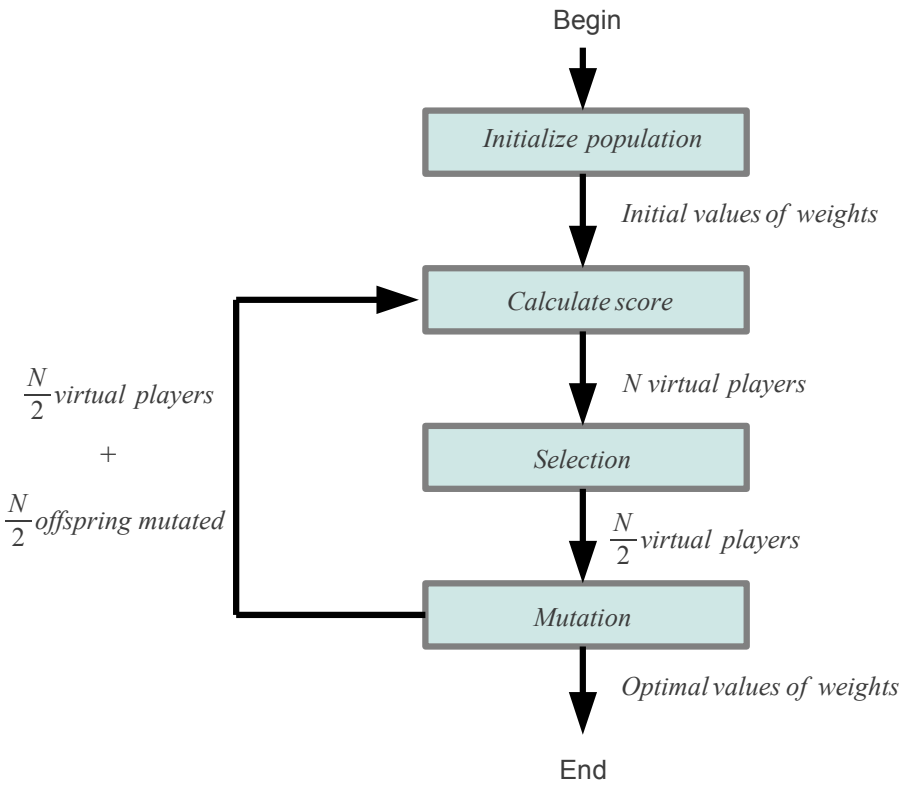

Figure 37: Flowchart of our proposed evolutionary algorithm.

can be performed, but this is not particularly relevant at this point, since our main aim here is to present some proof-of-principle results of our proposed methodology.

5.4 EXPERIMENTAL RESULTS

5.4.1 *Tuning weights*

In our experiments, we tuned the weights of the pieces and their mobility as shown in eq. (18). The population size N was set to 10, and the number of training games P was set to 6. Initialization took place using randomly generated values within the vicinity of their "theoretical" values ($\pm$200 points).

The "theoretical" values of the pieces are: 300, 300, 500 and 900 for the knight, bishop, rook and queen, respectively. The "theoretical" value of the mobility weight is 10, and its bounds are $[0, 300]$. The "theoretical" values of the pieces were discussed in Section 4.2.1 in the page 64. 30 runs were carried out under these conditions, and in all of them, the "theoretical" values were reached for all pieces.

In order to visualize better the convergence process, we carried out an additional run (number 31) in which the material values were generated within the range $[400, 500]$ and the mobility weight was set in the interval $[0, 300]$. At generation 0 for this run, the average weight values and their standard deviations are shown in Table 12.

Weight	**Value**	**Standard deviation**
X_{pawn}	100.00	0.00
X_{knight}	499.42	34.98
X_{bishop}	464.60	88.67
X_{rook}	469.85	122.75
X_{queen}	437.57	85.90
$X_{mobility}$	97.19	173.67

Tabla 12: Average weight values and their standard deviations for run number 31 (generation 0)

At the end of run 31, and after 50 generations, the average weight values and their standard deviations are shown in Table 13.

The average weight values and their standard deviations for 50 generations are shown in Figs. 38 and 39, respectively.

From the obtained results, we can see that the tuning process after 50 generations resulted in standard deviation values which are lower than those reported by [6] and [68].

The computational time required by our proposed approach to run during 50 generations was 3 minutes with 34 seconds under the operating system openSuse, using a PC with a 64 bits architecture, having two cores running at 2.8 Ghz. Unfortunately, most of the references that we consulted do not report

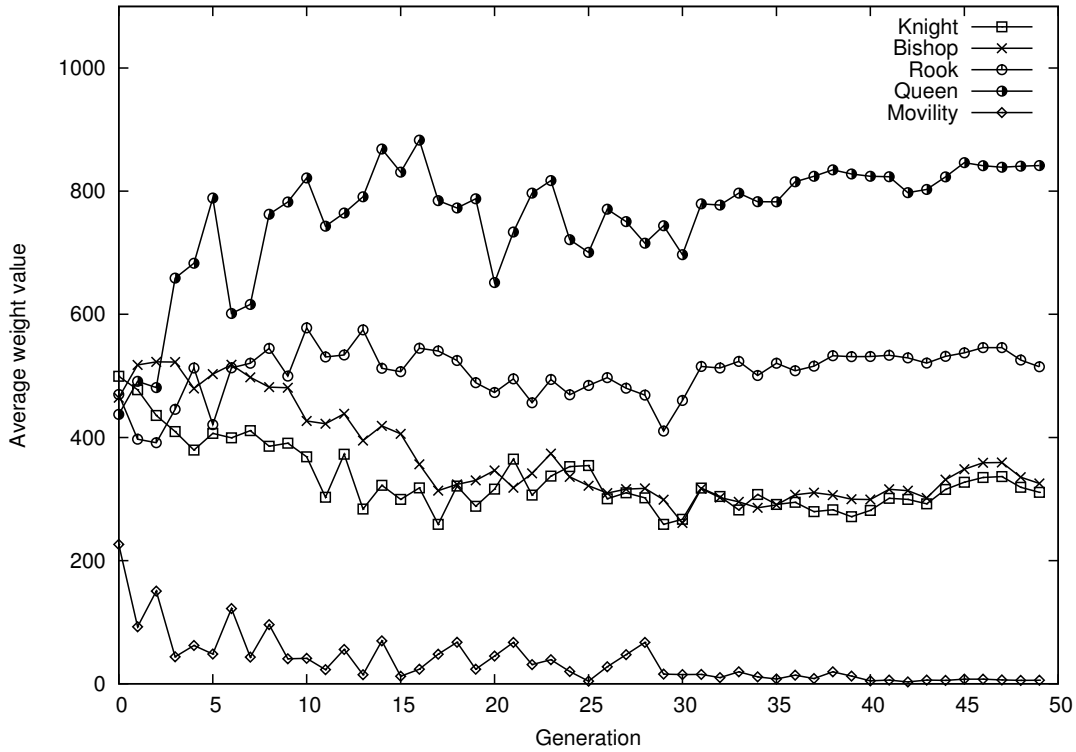

Figure 38: Average weight values of the population during 50 generations.

any CPU times that could give us an idea of the efficiency of our approach. The only reference in which we found such information is [30], in which Fogel et al. reported using a 2.2-Ghz Celeron PC with 128 MB of RAM. His program required 36 hours for executing 50 generations. However, it is important to indicate that he optimized many more weights that our approach (namely, the weights of three neural networks, the weights of the positional values, etc., see Section 3.4 in the page 50) and adopted a search depth of 4 ply. Thus, this execution time is not comparable with ours and is provided here just as a reference.

It is also worth indicating that the CPU time required by our proposed approach depends on the number of games that are randomly chosen from the database to compute the score of a virtual player during a generation of our evolutionary algorithm. In our case, we adopted $P = 6$.

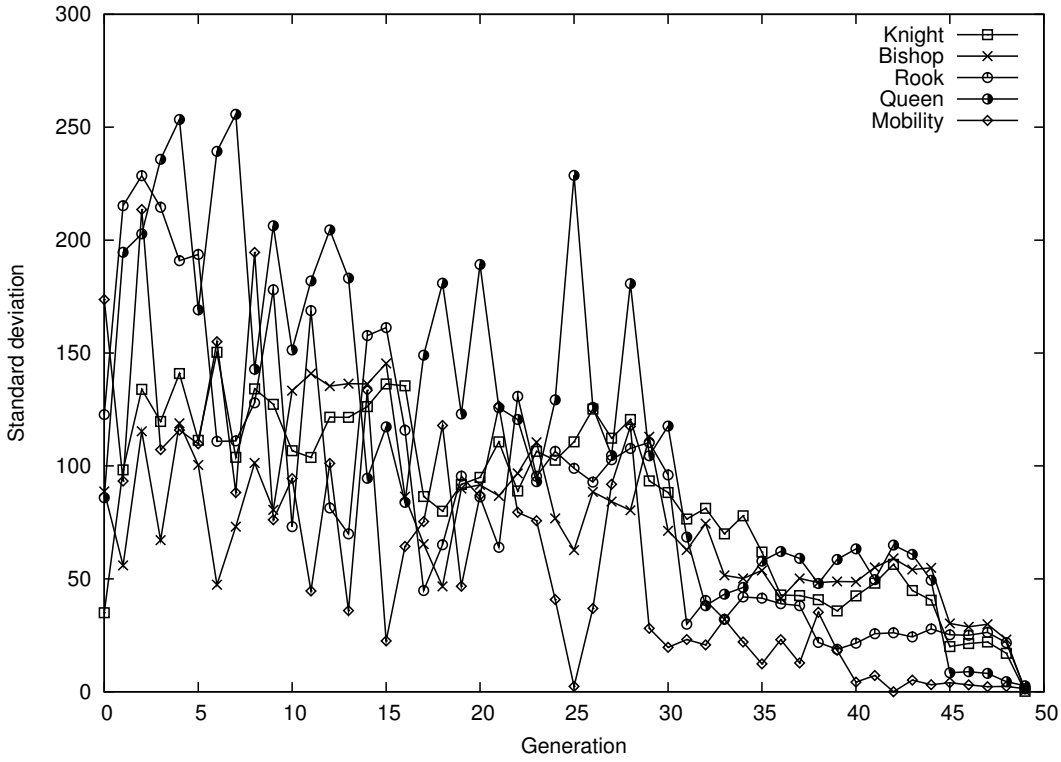

Figure 39: Standard deviation of the weights in the population during 50 generations.

5.4.2 *Additional Games*

We also performed an additional experiment. We carried out 100 games in which the first virtual player adopted the average weights from generation 0 of our evolutionary algorithm and the second virtual player adopted the average weights from generation 50. The scores achieved by them were 84 from the second player (who used the weights from generation 50) versus 16 from the first player. Next, we show one of the games in which the second virtual player defeated the first virtual player.

[**White: "Average weights in generation 0"**]
[**Black: "Average weights in generation 50"**]
[**Result: "0-1"**]
1 d4 d5 2 c4 c5 3 ♘c3 ♘f6 4 d×c5 d4 5 ♘b1 ♘c6 6 e3 e5 7 e×d4 e×d4 8 ♘f3 ♗×c5 9 ♗d3 O-O 10 ♗g5 ♖e8+ 11 ♗e2 ♕e7 12 ♗×f6 g×f6 13 a3 a5 14 a4 ♗f5 15 ♘a3 d3 16 ♘h4 ♕×e2+ 17 ♕×e2 ♖×e2+ 18 ♔d1 ♗g6 19 ♘×g6 f×g6 20 ♖f1

Weight	Value	Standard deviation
X_{pawn}	100.00	0.00
X_{knight}	310.89	0.22
X_{bishop}	325.32	0.45
X_{rook}	514.92	1.26
X_{queen}	841.61	2.62
$X_{mobility}$	5.62	1.34

Tabla 13: Average weight values and their standard deviations for run number 31 (generation 50)

♖ae8 21 ♘b5 ♗×f2 22 ♘c3 ♖e1+ 23 ♖×e1 ♖×e1+ 24 ♔d2 ♖×a1 25 ♔×d3 ♘e5+ 26 ♔e2 ♗d4 27 ♘d5 ♘×c4 28 ♘×f6+ ♗×f6 29 ♔d3 ♘×b2+ 30 ♔c2 ♖×a4 31 g3 ♖c4+ 32 ♔b3 ♔f7 33 h4 b5 34 h5 g×h5 35 g4 h×g4 36 ♔a3 g3 37 ♔b3 g2 38 ♔a3 g1♕ 39 ♔a2 ♘d3 40 ♔b3 ♕b1+ 41 ♔a3 ♕b2 ++

We also carried out 100 games between the best virtual player in generation 0 versus the best virtual player in generation 50, with a score of 85 to 15 in favor of the second virtual player. Next, we show one of the games in which the second virtual player defeated the first virtual player.

[**White: "The best virtual player in generation 0"**]
[**Black: "The best virtual player in generation** 50**"**]
[**Result: "0-1"**]
1 ♘f3 d5 2 d4 ♘f6 3 c4 c6 4 ♘c3 e6 5 c5 ♘e4 6 ♘b1 ♗e7 7 b4 ♕c7 8 g3 g5 9 ♕d3 ♘d7 10 ♗h3 h5 11 ♕×e4 d×e4 12 ♘g1 a5 13 ♗g2 f5 14 b×a5 ♕×a5+ 15 ♗d2 ♕a4 16 ♗c3 ♗d8 17 ♗b2 ♕c2 18 ♗a3 ♗a5+ 19 ♘c3 ♕×c3+ 20 ♔f1 ♕×a1+ 21 ♗c1 ♕×c1++

Additionally, we carried out 10 games between a virtual player which adopted the average weights from generation 50 and a (human) player ranked at 1600 points. The result was 9 to 1 in favor of the human player. Based on these

played games, we used the Bayeselo tool[1] to estimate the ratings for both the human player and the chess engine using a minorization-maximization algorithm [51]. The obtained ratings are shown in Table 18. The description of the columns in this table were given in Section 4.5.2 in the page 78. In this table, we can see that the rating obtained for the human player was 1737 and for the chess engine was 1463. Next, we can see the game that was won by the virtual player.

[**White: "Human player"**]
[**Black: "Average weights in generation 50"**]
[**Result: "0-1"**]
1 c4 e5 2 ♘c3 ♘f6 3 e4 ♗b4 4 ♘ge2 O-O 5 h3 c6 6 a3 ♗×c3 7 ♘×c3 d5 8 c×d5 c×d5 9 e×d5 e4 10 g3 ♗f5 11 ♗g2 ♕c8 12 f3 e×f3 13 ♕×f3 ♖e8+ 14 ♘e2 ♗e4 15 ♕f2 ♗×g2 16 ♕×g2 ♘bd7 17 O-O ♘c5 18 d4 ♘cd7 19 ♗g5 h6 20 ♗×f6 ♘×f6 21 ♖ac1 ♕d7 22 ♘f4 b6 23 ♖f2 ♖ad8 24 ♖cf1 ♖e7 25 b4 ♘×d5 26 ♘h5 ♘e3 27 ♕f3 ♘×f1 28 ♖×f1 ♕×d4+ 29 ♔h2 ♕e3 30 h4 ♖d2+ 31 ♔h3 ♕×f3 32 ♖×f3 g6 33 ♘f6+ ♔g7 34 ♘g4 h5 35 ♘h2 ♖ee2 36 ♘f1 ♖a2 37 g4 ♖f2 38 ♖×f2 ♖×f2 39 ♘h2 h×g4+ 40 ♘×g4 ♖f3+ 41 ♔g2 ♖×a3 42 ♘e5 ♖a4 43 ♘c6 ♔h6 44 ♔g3 ♔h5 45 b5 f5 46 ♘e5 a5 47 ♘d7 ♖b4 48 ♘×b6 ♖×b5 49 ♘c4 a4 50 ♘a3 ♖b3+

White resigns (see the final position in Figure 40).

It is worth indicating that in these games both virtual players used a database for openings and the depth of the search was set to 4 ply.

Rank	Name	Elo	+	-	Games	Score (%)	Opposition	Draws (%)
1	Human player	1737	132	92	10	90%	1463	0%
2	Chess engine	1463	92	132	10	10%	1737	0%

Tabla 14: Ratings for the human player and our chess engine in a ten-game match. The final result was 9 to 1 for the human player.

1 http://remi.coulom.free.fr/Bayesian-Elo/

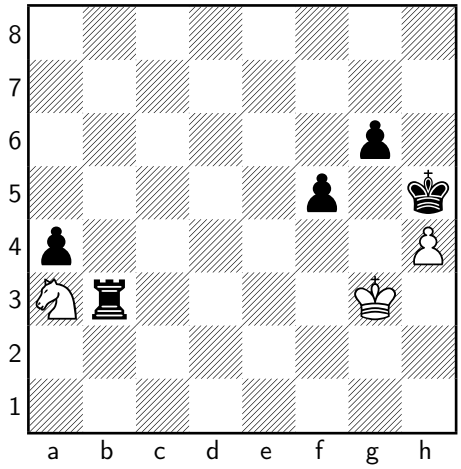

Figure 40: Final position for the game between the human player ranked at 1600 points (with white pieces) versus "average weights in generation 50" (with black pieces).

5.5 FINAL REMARKS OF THIS CHAPTER

We have reported here an evolutionary algorithm which incorporates a selection mechanism that favors virtual players that are able to "visualize" (or match) more moves from those registered in a database of chess grandmaster games. This information is used to tune the weights of our evaluation function, which is relatively simple to implement.

Our results indicate that the weight values obtained by our proposed approach closely match the known values from chess theory. These results give confidence in our method.

In the next chapter, and aiming to create a chess program that will be able to play at the level of a master or a chess master level, we plan to tune more weights (e.g., king security, doubled pawns, isolated pawns, passed pawns, rooks in open columns, rooks in seventh row, control center of the board, and so on) using our proposed evolutionary algorithm. We plan to use local search strategies to carry out a more fine-grained tuning of the weights. Our aim is to increase the rating of our chess engine as much as we can, adopting relatively inexpensive approaches (computationally speaking).

"Chess is war over the board. The object
is to crush the opponents mind."
Bobby Fischer

6

TUNING WEIGHTS WITH THE HOOKE-JEEVES METHOD

In Chapter 5, we implemented a chess engine based on evolutionary programming with a selection mechanism relying on grandmaster's chess games. The objective was to decide the virtual players that would pass to the following generation. Here, we use these same techniques to adjust a larger number of weights (twenty eight in this work against the five used in Chapter 5). The aim is to improve the rating of our chess engine. We also introduce here the use of a local search scheme based on the Hooke-Jeeves algorithm [50], which is adopted to adjust the weights of the best virtual player obtained in the evolutionary process. This approach produced an improvement of more than 220 points the chess rating system. As in Chapter 5, the chess pieces' material values for the best organism produced by the evolutionary process of this chapter are close to the "theoretical" values.

In the approach of this chapter we used 28 weights and we obtained a rating of 2424 points. In contrast, the neural network architecture proposed in Chapter 4 used 426 weights and our chess engine registered 2178 rating points. Therefore, this method represents an increase of 246 rating points and a reduction of 93.4% in the weights number with respect to the method of Chapter 4.

6.1 EVALUATION FUNCTION

A *factor* is a positional characteristic of a particular piece. Among the evaluation factors used are mobility, column type, and so on. A selection of these

factors was done for each piece, and they are multiplied by weights to change their influence in the evaluation function. These weights are tuned by an evolutionary process in order to obtain better virtual players.

For carrying out the experiments of this chapter, we used the chess engine described in Section 2.7 in the page 26. The only difference is the type of evaluation function. Now, our chess program evaluates board positions from a particular side with:

$$eval = pV + mV, \tag{21}$$

where mV (pV) is the sum of material values (positional values) for the side under evaluation. In Sections 4.2.1 and 4.2.2 in the page 64 are the definitions of material and positional values, respectively. We defined mV and pV, respectively by

$$mV = \sum_{i=1}^{r} X_i, \tag{22}$$

$$pV = \sum_{i=1}^{r} P_i, \tag{23}$$

here X_i, and P_i represent the material and positional values for piece i, respectively, and r is the number of pieces. All these parameters are measured with respect to the side under evaluation. In equation 22 the king's material value should not be taken into account.
Next, we will describe how to obtain the positional values of the chess pieces.

6.1.1 *King's positional value*

The king's positional value is given by:

$$P_{king} = \sum_{i=1}^{4} X_{king,i} * F_{king,i}, \tag{24}$$

here $X_{king,i}$ is the weight factor of $F_{king,i}$.

The king's factors are the input signals of the king's neural network in Section 4.3.1.1 in the page 66, namely:

1. $F_{king,1}$ is equal to *Defending material* input signal.
2. $F_{king,2}$ is equal to *Attacking material* input signal.
3. $F_{king,3}$ is equal to *Castling* input signal.
4. $F_{king,4}$ is equal to *Pawns* input signal.

6.1.2 *Queen's positional value*

The queen's positional value is given by:

$$P_{queen} = X_{queen,1} * F_{queen,1}, \tag{25}$$

here $X_{queen,1}$ is the weight factor of $F_{queen,1}$.

At the moment, we only considered one factor:

1. $F_{queen,1}$ is the queen's mobility.

In Section 4.3.1.2 in the page 68 the queen's neural network used four input signals to obtain its positional value. Now, we used only one factor, and we hope that by adding the remaining input signals (*Column type*, *Row* and *Column*) our chess engine could increase its rating with respect to the value reported in this chapter.

6.1.3 *Rook's positional value*

The rook's positional value is given by:

$$P_{rook} = \sum_{i=1}^{7} X_{rook,i} * F_{rook,i}, \tag{26}$$

here $X_{rook,i}$ is the weight factor of $F_{rook,i}$.

The rook's factors are the following:

1. $F_{rook,1}$ is equal to the input signal *Rook mobility* in Section 4.3.1.3 in the page 70.

2. $F_{rook,2}$ is true if on the rook's column there are no pawns; otherwise, it is false.

3. $F_{rook,3}$ is true if on the rook's column there are only adversary pawns; otherwise, it is false.

4. $F_{rook,4}$ is true if on the rook's column there are pawns for both sides and the rook is in front of its pawns; otherwise, it is false.

5. $F_{rook,5}$ is true if on the rook's column there are pawns on both sides and the rook is behind its pawns; otherwise, it is false.

6. $F_{rook,6}$ is equal to the input signal *Seventh row* in Section 4.3.1.3.

7. $F_{rook,7}$ is equal to the input signal *Seventh row folded* in Section 4.3.1.3.

We tested all possible subsets of the features reported in Section 4.3.1.3 in the page 70. The subset used here is the one producing the best results. In this way, the input *Column type* in Section 4.3.1.3 was replaced by factors 2, 3, 4, and 5.

6.1.4 *Bishop's positional value*

The bishop's positional value is given by:

$$P_{bishop} = X_{bishop,1} * F_{bishop,1}, \tag{27}$$

here $X_{bishop,1}$ is the weight factor of $F_{bishop,1}$.

At the moment, we only considered one factor:

1. $F_{bishop,1}$ is the bishop's mobility.

In Section 4.3.1.4 in the page 71 the bishop's neural network used four input signals to obtain its positional value. Now, we used only one factor, and we hope that by adding the remaining input signals (*Pawn's mobility*, *Ahead* and *Weight*) our chess engine could increase its rating with respect to the value reported in this chapter.

6.1.5 *Knight's positional value*

The knight's positional value is given by:

$$P_{knight} = \sum_{i=1}^{7} X_{knight,i} * F_{knight,i}, \tag{28}$$

here $X_{knight,i}$ is the weight factor of $F_{knight,i}$.

The knight's factors are the following:

1. $F_{knight,1}$ is equal to the input signal *Knight mobility* in Section 4.3.1.5 in the page 72.
2. $F_{knight,2}$ is equal to the input signal *Supported* in Section 4.3.1.5.
3. $F_{knight,3}$ is equal to the input signal *Operation base* in Section 4.3.1.5.
4. $F_{knight,4}$ is true if the knight is in the squares $a_1, \ldots, a_8$, $b_1, \ldots, g_1$, $h_1, \ldots, h_8$, and $b_8, \ldots, g_8$ (which corresponds to the squares on the periphery of the board); otherwise, it is false.
5. $F_{knight,5}$ is true if the knight is in the squares $b_2, \ldots, b_7$, $c_2, \ldots, f_2$, $g_2, \ldots, g_7$, and $c_7, \ldots, f_7$; otherwise, it is false.
6. $F_{knight,6}$ is true if the knight is in the squares $c_3, \ldots, c_6$, d_3, e_3, $f_3, \ldots, f_6$, and $d_6, \ldots, e_6$; otherwise, it is false.
7. $F_{knight,7}$ is true if the knight is in the squares d_4, e_4, d_5, e_5; otherwise, it is false.

We tested all possible subsets of the features reported in Section 4.3.1.5. The subset used here is the one producing the best results. In this way, the input *Periphery* in Section 4.3.1.5 was replaced by factors 4, 5, 6, and 7.

We expected that $X_{knight,4} < X_{knight,5} < X_{knight,6} < X_{knight,7}$ because if the knight is located in the center of the board its positional value will be better.

6.1.6 *Pawn's positional value*

The pawn's positional value is given by:

$$P_{pawn} = \sum_{i=1}^{5} X_{pawn,i} * F_{pawn,i}, \tag{29}$$

here $X_{pawn,i}$ is the weight factor of $F_{pawn,i}$.

The pawn's factors are the following:

1. $F_{pawn,1}$ is equal to the input signal *Doubled* in Section 4.3.1.6 in the page 73.
2. $F_{pawn,2}$ is equal to the input signal *Isolated* in Section 4.3.1.6.
3. $F_{pawn,3}$ is equal to the input signal *Central* in Section 4.3.1.6.
4. $F_{pawn,4}$ is equal to the input signal *Past* in Section 4.3.1.6.

6.2 METHODOLOGY

Our purpose now is to tune the weights of Section 6.1 using evolutionary programming. We proceed in two phases. In the first one, we use the algorithm in Chapter 5 to adjust the weights in equations (22) to (29). In the second phase, a local search based on the Hooke-Jeeves algorithm is used to look for local improvements in the optimal organism found. The aim is that the weights adjustment performed by our approach increase in the rating of the chess engine.

First, we give a structural description of the evolutionary algorithm.

6.2.1 *Components of our evolutionary algorithm*

The main components of our evolutionary algorithm were described in Section 5.3.1 in the page 87. Here, we change the composition of the chromosome and the population size. Now, the chromosome encodes the twenty weights in equations (22) to (29). These weights are shown in Table 15 together with the material value of the pawn. As in Chapter 5, we assign to each pawn a material value of 100 points and consider this as the reference value for the other

pieces. Also, we changed the population size N from 10 to 20 individuals due to the fact that in the experiments the rating recorded by our chess engine was better for the last value.

6.2.2 *Phases of our method*

The method consists of the following phases or steps:

- **Exploration search**. It is the first step of the method, and is based on evolutionary programming [33] which has a selection mechanism based on a database of chess grandmaster games (supervised learning). The selection mechanism allows that the virtual players with more positions properly solved from a database of chess grandmaster games acquire the right to pass to the next generation. In Chapter 5, we conducted this phase in a similar way, but now we adjusted a larger number of weights (we went from five to twenty eight).
- **Exploitation search**. In this second step of the method, we carried out a local search procedure, aiming to improve the best virtual player obtained in the previous step. For that sake, we applied the Hooke-Jeeves algorithm to the best virtual player obtained in the exploration search. The objective function incorporated into the Hooke-Jeeves method also used a database of chess grandmaster games to carry out the weights adjustment under consideration.

Algorithm 10 EvolutionaryAlgorithm()

```
1: intializePopulation();
2: g = 0;
3: while g++< Gmax do
4:    scoreCalculation();
5:    selection();
6:    mutate();
7:    g++;
8: end while
```

In Section 5.3.2 in the page 88, we showed the flowchart of the evolutionary algorithm applied to the exploration search. Now, we show the version of the evolutionary algorithm for this flowchart diagram in Algorithm 10. The evolutionary algorithm and the flowchart operate identically. Its description is as

follows. Line 1 initializes the weights of N virtual players with random values within their corresponding boundaries. Line 2 sets the generations counter equal to zero. Lines 3 to 8 carry out the weights adjustment for virtual players during $Gmax$ generations. In line 4, we calculate the score for each virtual player (Algorithm 9 in Section 5.3.2 in the page 88 describes in more detail this aspect). In Line 5 we apply the selection mechanism so that only the best $N/2$ virtual players pass to the following generation. In line 6, we mutate the first half of the population in order to obtain the second half of the virtual players. That is, all weights from each surviving parent were mutated to create one offspring (the weights that were mutated are shown in Table 15). As it was done in Chapter 5, we adopted here Michalewicz's non-uniform mutation operator [66]. Since, we adopted evolutionary programming, no crossover operator is employed in our case. Finally, line 7 increases the generation counter by 1.

The procedure for computing the score of each virtual player is described in Algorithm 9 of Section 5.3.2 in the page 88.

In the exploitation search, we employed the Hooke-Jeeves method to further adjust the weights of the best virtual player obtained during the exploration search step. The Hooke-Jeeves method is a direct search algorithm originally proposed in 1961 [50]. This method carries out a deterministic local search with a local descent algorithm, which does not make use of the objective function derivatives.

The fundamental part of the Hooke-Jeeves method is shown in Algorithm 11. Basically, this method starts in an initial base point, $\mathbf{x}_0$, and explores each coordinate axis with its own step size through the function explore($\mathbf{x}_0$, $\mathbf{h}$) describe in Algorithm 12, where h is the vector of increase in each coordinate axis. Trial points in all D coordinate directions are compared until the best point, $\mathbf{x}_1$, is found. If the best new trial point is better than the base point, then another attempt to obtain another move in the same direction is made. If none of the trial points improve the solution $\mathbf{x}_0$, the step is presumed to have been too large, so the procedure repeats with smaller step sizes.

The objective function f returns the number of positions solved by each virtual player. In this case, we randomly chose $M = 20$ positions from chess grandmaster games that were not solved by any virtual player during the exploratory search.

Algorithm 11 HookeJeeves()

{as long as step length is still not small enough}
while $h > h_{min}$ **do**
  {explore the parameter space}
  x_1 = explore(x_0, h);
  {if improvement could be made}
  if $f(x_1) < f(x_0)$ **then**
    {make differential pattern move}
    $x_2 = x_1 + (x_1 - x_0)$;
    if $f(x_2) < f(x_1)$ **then**
      $x_0 = x_2$;
    else
      $x_0 = x_1$;
    end if
  else
    $h = h *$ reduction_factor;
  end if
end while

Algorithm 12 explore(vector x_0, vector h)

{e_i is the unit vector for coordinate i}
{for all D dimensions}
for $i = 0$; $i < D$; $i++$ **do**
  {check coordinate i}
  if $f(x_0 + e_i * h) < f(x_0)$ **then**
    $x_0 = x_0 + e_i * h$;
  else if $f(x_0 - e_i * h) < f(x_0)$ **then**
    $x_0 = x_0 - e_i * h$;
  end if
end for
return x_0;

6.2.3 *Initialization*

During the exploratory search step, the initial population consisted of N = 20 virtual players (10 parents and 10 offspring in subsequent generations). Their weights (described in equations (22), (24), (25), (26), (27), (28) and (29)) were randomly generated with a uniform distribution within their allowable bounds (these bounds for each weight are shown in Table 15). These allowable bounds were taken from other related works (see for example, Bošković et al. [5]).

6.2.4 *Database of games*

In the experiments reported in this chapter, we used a database consisting of 1000 games from chess grandmasters having a rating above 2600 Elo points (see Appendix A). The games were taken from the Linares tournaments, from matches for the world chess championship, and from the Wijk aan Zee tournaments, among others. These games can be download from: `http://www.chessbase.com/`.

6.3 EXPERIMENTAL RESULTS

We carried out three experiments. The first experiment was based on the exploration and the exploitation stages of the search. In the second experiment, we performed matches between the best virtual player obtained after the exploration phase and the best virtual player obtained after the exploitation phase. Finally, in the third experiment, we carried out matches between virtual players and the popular chess program *Chessmaster*.

In these experiments, our chess engine used the database *Olympiad.abk* in the opening phase. This database is included with the graphical user interface *Arena*[1]. In the following sub-sections, we will describe these experiments.

6.3.1 *First experiment*

The first experiment was divided into two steps described in Section 6.2. In the first step, we applied exploration search to adjust the weights shown in Table 15. In this case, we performed 30 runs, and in each of them, we used

1 http://www.playwitharena.com/

$Weight$	W_{low}	W_{high}
X_1 (PAWN_VALUE)	100	100
X_2 (KNIGHT_VALUE)	200	400
X_3 (BISHOP_VALUE)	200	400
X_4 (ROOK_VALUE)	400	600
X_5 (QUEEN_VALUE)	800	1000
$X_{king,1}$	0	4000
$X_{king,2}$	−4000	0
$X_{king,3}$	0	100
$X_{king,4}$	0	100
$X_{queen,1}$	0	100
$X_{rook,1}$	0	100
$X_{rook,2}$	−50	50
$X_{rook,3}$	−50	50
$X_{rook,4}$	−50	50
$X_{rook,5}$	−50	50
$X_{rook,6}$	0	100
$X_{rook,7}$	0	100
$X_{bishop,1}$	0	100
$X_{knight,1}$	0	100
$X_{knight,2}$	0	100
$X_{knight,3}$	0	100
$X_{knight,4}$	−50	50
$X_{knight,5}$	−50	50
$X_{knight,6}$	−50	50
$X_{knight,7}$	−50	50
$X_{pawn,1}$	−50	50
$X_{pawn,2}$	−50	50
$X_{pawn,3}$	−50	50
$X_{pawn,4}$	−50	100

Tabla 15: Ranges of the weights for each virual player.

$Gmax = 200$ generations, $N = 20$ virtual players, and $p = 1000$ training positions for chess grandmaster games. The best virtual player from these runs at generation 0, and at generation 200, were called $VP^{0}_{exploration}$, and $VP^{200}_{exploration}$, respectively.

Figure 41 shows the evolutionary process for the exploration search. The plot shows the number of positions solved (a total of 1000) for the best virtual player and the average weight values of 20 virtual players during 200 generations. At generation 0, the number of positions solved for the average weight values was 187 (which corresponds to 18.7% of the positions), and 208 for the best virtual player (which corresponds to 20.8% of the positions). At generation 200, the number of positions solved for the average weight values and the best virtual player was 328 (which corresponds to 32.8% of the positions). Note that this value is competitive with the value reported in [21], which corresponds to 32.4% of the positions. At generation 200, the number of positions solved for the average weight values and the best virtual player was the same because we used Michalewicz's non-uniform mutation operator (see Section 5.3.1 in the page 87).

In the second step of the first experiment, we applied the exploitation search to the best virtual player obtained with the exploration search. In this case, we applied the Hooke-Jeeves algorithm with the following parameters:

- The step reduction factor $\alpha = 2$.
- The termination parameter $\epsilon = 0.5$.
- The increments $\Delta_i = 30$ for $i = 1, \ldots, W_N$ (W_N is the number of weights).

The second column in Table 16 shows the weights tuning for the virtual player $VP^{200}_{exploration}$. In the third column, we show the weights tuning for the virtual player $VP_{exploitation}$. In both cases, we can see that the material values of pieces are close to their "theoretical" values.

Next, we used the resulting weights of the virtual player $VP_{exploitation}$ to test them with 1000 training positions from chess grandmaster games. In this case, the virtual player $VP_{exploitation}$ successfully resolved 483 of the 1000 positions (which corresponds to 48.3%). Therefore, we can see that the number of positions solved using both exploration and exploitation was larger than when we used only exploration (from 32.8% to 48.3%). We consider that the number of positions solved by this method is satisfactory because we used only a depth of one ply in the search tree. It is noteworthy that David-Tabibi et al. [21] also used one ply in their work.

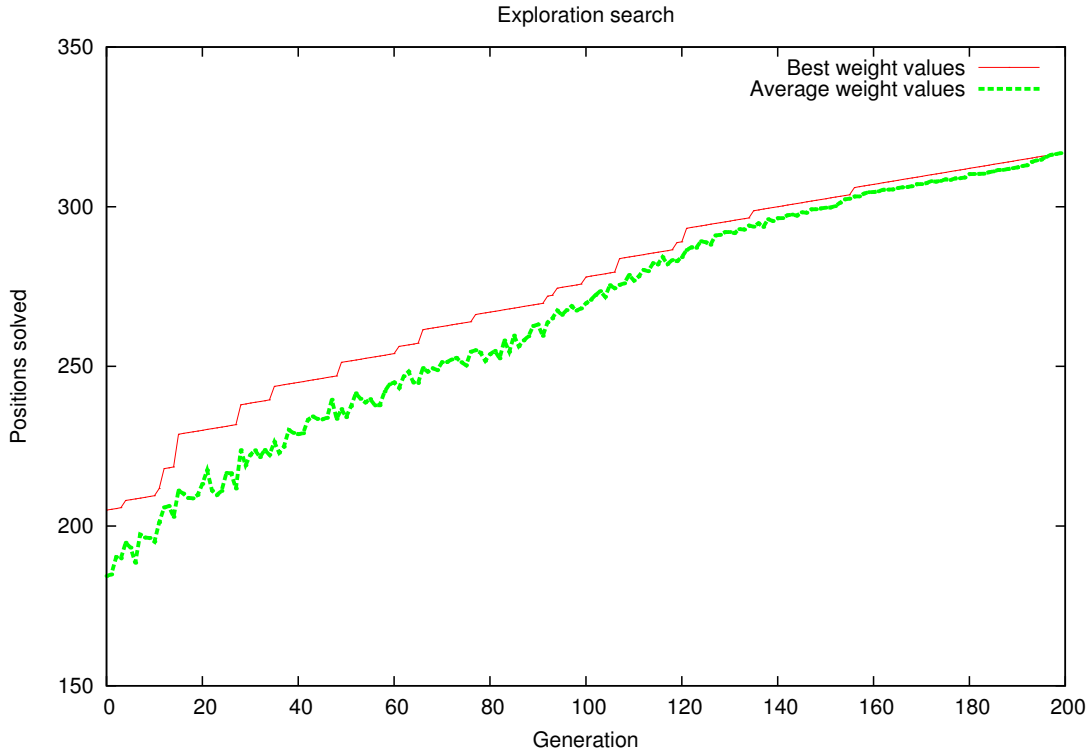

Figure 41: Evolutionary process for the exploration search. The plot shows the number of positions solved (a total of 1000) for the best virtual player and the average weight values of the 20 virtual players during 200 generations.

With the completion of the exploration and exploitation search, we used 1000 additional positions for testing the virtual player $VP_{exploitation}$. We let this virtual player perform a one ply search on each of these positions, and the percentage of correctly solved positions was 47.9%. This indicates that the first 1000 positions used for training, cover most of the types of positions that can arise.

6.3.2 *Second experiment*

In this experiment, our chess engine used a search depth of four plies. We carried out 200 games between the virtual player $VP^{200}_{exploration}$ and the virtual player $VP_{exploitation}$ (each virtual player played 100 games with black pieces and 100 games with white pieces). The virtual player $VP_{exploitation}$ won,

$Weight$	$VP^{200}_{exploration}$	$VP_{exploitation}$
X_1 (PAWN_VALUE)	100	100
X_2 (KNIGHT_VALUE)	297	302
X_3 (BISHOP_VALUE)	315	319
X_4 (ROOK_VALUE)	502	506
X_5 (QUEEN_VALUE)	923	910
$X_{king,1}$	1650	1675
$X_{king,2}$	−1430	−1425
$X_{king,3}$	47	45
$X_{king,4}$	65	72
$X_{queen,1}$	5	9
$X_{rook,1}$	62	73
$X_{rook,2}$	45	46
$X_{rook,3}$	27	33
$X_{rook,4}$	32	27
$X_{rook,5}$	−8	−7
$X_{rook,6}$	63	68
$X_{rook,7}$	78	82
$X_{bishop,1}$	72	76
$X_{knight,1}$	64	68
$X_{knight,2}$	56	72
$X_{knight,3}$	52	73
$X_{knight,4}$	−12	−15
$X_{knight,5}$	3	6
$X_{knight,6}$	15	26
$X_{knight,7}$	42	43
$X_{pawn,1}$	−32	−44
$X_{pawn,2}$	−47	−48
$X_{pawn,3}$	−44	−41
$X_{pawn,4}$	43	48

Tabla 16: Values of the weights after the exploration search (shown in the second column) and after the exploitation search (shown in the third column).

drew, and lost 142, 25, and 32 games, respectively, versus the virtual player $VP^{200}_{exploration}$.

Next, we used the Bayeselo tool[2] to estimate the ratings of the chess engine using a minorization-maximization algorithm [51]. The obtained ratings are shown in Table 17. The description of the columns in this table were given in Section 4.5.2 in the page 78. In this table, we can see that the rating for the virtual player $VP_{exploitation}$ was 2425, and the rating for the virtual player $VP^{200}_{exploration}$ was 2205, representing an increase of 220 rating points between the virtual player obtained with exploration plus exploitation search and the virtual player obtained only with exploration search.

Virtual player	Elo	+	-	Games	Score (%)	Opposition
$VP_{exploitation}$	2425	25	24	200	77%	2205
$VP^{200}_{exploration}$	2205	24	25	200	23%	2425

Tabla 17: Ratings of the second experiment.

We can have an idea of the playing strength of virtual players using the classification of the United States Chess Federation. From Table 19 (Appendix A), we can see that the strength of the virtual player $VP^{200}_{exploration}$ (2205 rating points) is at the level of a master in chess, and the strength of the virtual player $VP_{exploitation}$ (2425 rating points) is at the level of a senior master in chess.

6.3.3 *Third experiment*

In this experiment, we carried out 200 games among the virtual players $VP^{0}_{exploration}$, $VP^{200}_{exploration}$, $VP_{exploitation}$, and the popular chess program *Chessmaster* (grandmaster edition) which was set at 2500 rating points. The results are shown in Figure 42. In this figure, we can see that $Chessmaster_{2500}$'s wins, draws, and losses were 68, 118, and 14, respectively, versus the virtual player $VP_{exploitation}$ (denoted as the histogram H1 in Figure 42). Also, $Chessmaster_{2500}$'s wins, draws, and losses, were 158, 28, and 14, respectively, versus the virtual player $VP^{200}_{exploration}$ (denoted as the histogram H2 in Figure 42), respectively, and so on.

Based on these played games, we used again the Bayeselo tool to estimate the ratings of $Chessmaster_{2500}$, $VP_{exploitation}$, $VP^{200}_{exploration}$, and $VP^{0}_{exploration}$.

2 http://remi.coulom.free.fr/Bayesian-Elo/

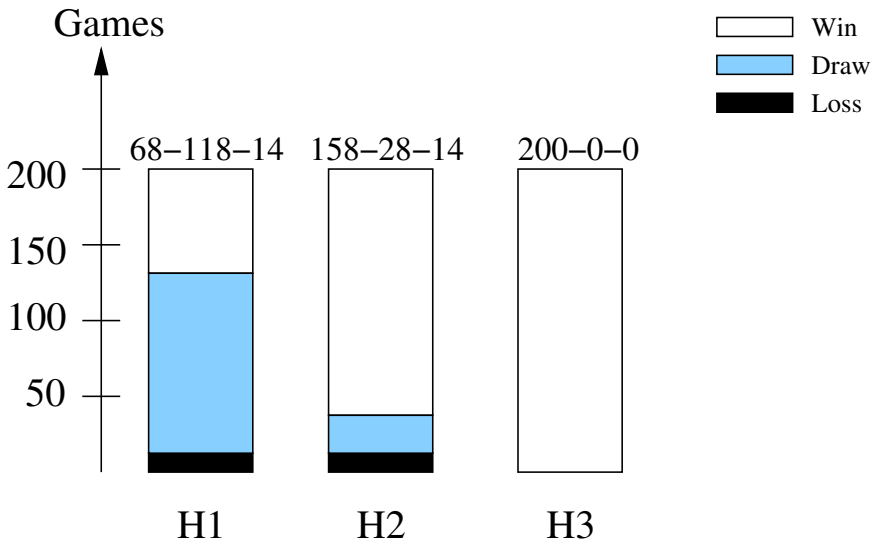

Figure 42: Histogram of wins, draws and losses for $Chessmaster_{2500}$ against $VP_{exploitation}$ (H1), $VP^{200}_{exploration}$ (H2), $VP^{0}_{exploration}$ (H3).

The obtained ratings are shown in Table 18. The description of the columns in this table were given in Section 4.5.2 in the page 78. In this table we can see that the rating for the virtual players $VP^{0}_{exploration}$, $VP^{200}_{exploration}$, and $VP_{exploitation}$ were 1600, 2197, and 2424, respectively. In this experiment, the chess engine used a search depth of six plies.

Name	Elo	+	-	Games	Score (%)	Opposition	Draws (%)
$Chessmaster_{2500}$	2499	23	23	600	83%	2074	24%
$VP_{exploitation}$	2424	28	28	200	37%	2499	59%
$VP^{200}_{exploration}$	2197	39	43	200	14%	2499	14%
$VP^{0}_{exploration}$	1600	144	348	200	0%	2499	0%

Tabla 18: Ratings of the third experiment.

6.4 FINAL REMARKS OF THIS CHAPTER

The work of this chapter is an extension of the one presented in Chapter 5. In this case, we used two steps to carry out the weights tuning of a chess engine. In the first step, we performed an exploration search through an evolutionary

algorithm with supervised learning. One difference with respect to our previous work is that now we adjusted a larger number of weights (from five to twenty eight). With this change, we obtained an increase in the rating of the chess engine from 1463 to 2205 points.

Another difference with respect to Chapter 5 is the second step of this method. In this case, we used the Hooke-Jeeves algorithm to continue the weights adjustment for the best virtual player obtained in the previous step. Using this algorithm as a local search engine, we increased the rating of our chess engine from 2205 to 2425 points (in the second experiment), and from 2197 to 2424 points (in the third experiment).

7

CONCLUSIONS AND FUTURE WORK

In this work, we designed and implemented a computer chess program that plays chess, a chess engine, with the following features:

- Board representation through the 0×88 method.
- Election of movements through the alpha-beta algorithm.
- Stabilization of positions through the quiescence algorithm that takes into account the exchange of material and king's checks.
- Use of iterative deepening and hash tables.

We focused mainly in the incorporation of knowledge to the evaluation function through artificial intelligence techniques such as evolutionary algorithms and/or neural networks.

We proposed an original neural network architecture to obtain the positional values of chess pieces. We also used an evolutionary algorithm to tune the weights involved in these neural networks and the weights which obstruct the bishop's movement. The novelty of this architecture is that proposes an original method to obtain the positional values of chess pieces in a dynamic way depending on the characteristics of the position. With this proposal our chess engine reached a rating of 2,178 points. Although this value is really good (it corresponds to the level of an expert in chess by the United States Chess Federation), we are still relatively far from the main objective of this thesis, of obtain a 2600 points chess engine.

Due to the limitations in the rating obtained, we proceeded to investigate the exclusive use of evolutionary algorithms to carry out the weights adjustment of our chess engine. First, we used an evolutionary algorithm based on supervised learning to obtain the "theoretical values" of chess pieces and their mobility factor.

Finally, we used two steps to carry out the weights tuning of the chess engine. In the first step, we employed the previous evolutionary algorithm to adjust a larger number of weights (from five to twenty eight). With this change, we obtained an increase in the rating from 1463 to 2205 points. In the second step, we used the Hooke-Jeeves algorithm to continue the weights adjustment for the best virtual player obtained in the previous step. Using this method as a local search engine, we increased the rating of our chess engine from 2205 to 2425 points.

As part of our future work, and aiming to create a chess program that will be able to play at the level of 2600 rating points, we plan to do the following:

- Regarding the neural network architecture.
  - We can evolve the parameter c_i to obtain a more accurate relationship between the material and positional value of a piece. We think that this parameter is important to determine the style of play of our chess engine.
  - We plan to use better strategies that allow us a more efficient exploration of the search space in our neural network architecture.
  - We can evolve the topology of each neural network looking for its optimal configuration.
- Regarding the approach based on the Hooke-Jeeves method.
  - Adjust more weights, for example, the pawns weights that obstruct the bishop's movement, the queen's row, queen's column and queen's column type, among others.

Regarding to the two approaches we plan to carry out more experiments varying the population size and increasing the number of games in the database. We also plan to add extensions to the quiescence search such as pawn promotions. It is also desirable to add endgame databases to our chess engine.

A

ELO RATING SYSTEM

The **Elo rating system** is a method to represent the playing strength in two-player games such as chess, Go, association football, basketball, among others. This method was created by the mathematician Arpad Elo. This system has been adopted by the *United States Chess Federation* (USCF) since 1960 and by the *World Chess Federation* (FIDE, Fédération Internationale des Échecs, by its French acronym) since 1970. In Table 19, we show the classification of the USCF.

A.1 ELO FORMULA

The formula to obtain the Elo rating of a player is given by [26]:

$$R_{new} = R_{old} + K(outcome - W), \tag{30}$$

where:
R_{new} is the new rating.
R_{old} is the old rating.
K is a constant that depends on the rating.
$outcome$ is the game result.
W is the expected or percentage score given by the logistic curve.

The World Chess Federation uses the following values for the K-factor:

Interval	Level
2400 and above	Senior Master
2200 – 2399	Master
2000 – 2199	Expert
1800 – 1999	Class A
1600 – 1799	Class B
1400 – 1599	Class C
1200 – 1399	Class D
1000 – 1199	Class E

Tabla 19: Elo rating system

$$K = \begin{cases} 30, & \text{players with a smaller number of 30 games.} \\ 15, & \text{players with at least 30 games and less than 2400 rating points.} \\ 10, & \text{players with at least 30 games and more than 2400 rating points.} \end{cases}$$

The $outcome$ is given by:

$$outcome = \begin{cases} 1, & \text{for a win} \\ 0.5, & \text{for a draw} \\ 0, & \text{for a loss} \end{cases}$$

The expected or percentage score W is given by:

$$W = \frac{1}{1 + 10^{\frac{R_{opponent} - R_{old}}{400}}}, \tag{31}$$

here $R_{opponent}$ is the opponent's rating.

The rating difference is given by the term $R_{opponent} - R_{old}$. The Elo formula is a relationship between the rating difference and the expected score. In Table 43, we show some values for this relationship. This table is usually called *expectancy table* and in the page http://www.fide.com/fide/handbook.html?id=73&view=article we can see this relationship used by the World Chess Federation to calculate the rating of chess players. We can also see this theoretical relationship in Figure 43. After using the expected value in equation (30), we can know how the rating has changed for a particular player.

Rating difference	Expected score
0	0.5000
20	0.529
40	0.557
60	0.585
80	0.613
100	0.640
120	0.666
140	0.691
160	0.715
180	0.738
200	0.760
300	0.849
400	0.909

Tabla 20: Some values for the relationship between rating difference and expected score.

On the other hand, if we plot the *White rating minus the Black rating* against the *White's percentage score*, we obtain the Figure 44. In this figure, the square (□) denotes the White's percentage score for 266,000 games between 1994 and 2001, the white curve denotes the Elo's theoretical prediction and the blue line gives the best prediction. In this graph, we can see that the Elo rating system has a drawback. For example, in a game where two players have the same rating, the Elo rating system indicates a White's percentage score of 50%; however, the better prediction gives a percentage greater than this value [36]. In fact, this happens in this way because the white side moves first in chess.

There are two tools to estimate the ratings of chess players or chess engines: *Elostat* and *Bayeselo*. This tools can read a text file that contains the results of a set of games to compute the rating of the players involved. Elostat tool is included with the graphical user interface *Arena*[1], but it has the disadvantage of not taking into account the side's color to compute the rating involved.

1 http://www.playwitharena.com/

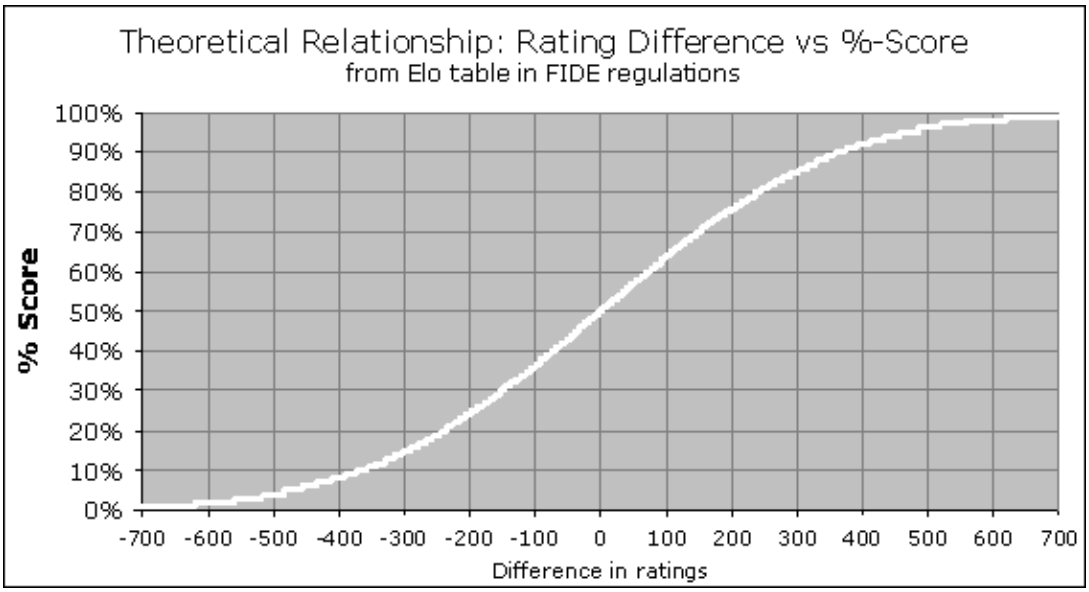

Figure 43: Rating difference versus percentage score. This figure was taken from http://www.chessbase.com/newsdetail.asp?newsid=7114.

Since the Bayeselo tool eliminates this small drawback, it was adopted to estimate the ratings in the experiments of Chapters 4, 5 and 6 of this thesis.

A.2 WORLD CHESS FEDERATION

As mentioned before, the World Chess Federation has adopted the Elo rating System. Until July 2012, the statics of the chess players with a rating higher than 2200 points are the following:

- 5839 players had a rating between [2200, 2299].
- 2998 players had a rating between [2300, 2399].
- 1382 players had a rating between [2400, 2499].
- 587 players had a rating between [2500, 2599].
- 178 players had a rating between [2600, 2699].
- 42 players had a rating between [2700, 2799].
- 5 active players had a rating over 2800 (Magnus Carlsen, Veselin Topalov, Viswanathan Anand, Vladimir Kramnik and Levon Aronian).

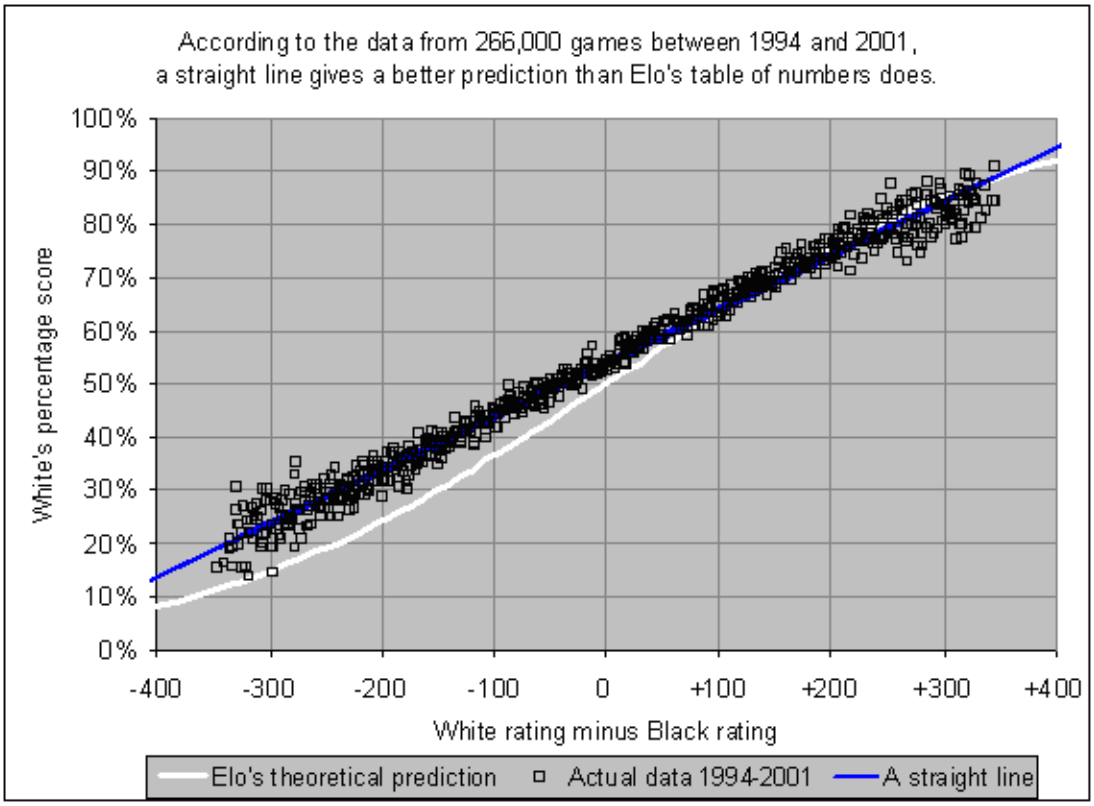

Figure 44: Comparison of the Elo's prediction and better prediction. This figure was taken from http://www.chessbase.com/newsdetail.asp?newsid=562.

The World Chess Federation grants four chess titles:

- *Candidate Master*. Chess players with a FIDE rating of at least 2200.
- *FIDE Master*. Chess players with a FIDE rating of at least 2300.
- *International Master*. Chess players with a FIDE rating of at least 2400.
- *Grandmaster*. Chess players with a FIDE rating of at least 2500.

In Table 21, we can see the classification of the World Chess Federation for the top ten chess players until October 2012. In Table 22, we can see some values for the number of chess grandmasters per country until July 2012.

Rank	Name	Country	Rating
1	Carlsen, Magnus	Norway	2843
2	Aronian, Levon	Armenia	2821
3	Kramnik, Vladimir	Russia	2795
4	Radjabov, Teimour	Azerbaijan	2792
5	Nakamura, Hikaru	United States	2786
6	Karjakin, Sergey	Russia	2780
7	Anand, Viswanathan	India	2780
8	Caruana, Fabiano	Italy	2772
9	Ivanchuck, Vassily	Ukraine	2771
10	Morozevich, Alexander	Russia	2758

Tabla 21: Top ten chess players until October 2012.

Rank	Country	Number of grandmasters
1	Russia	214
2	Ukraine	79
3	Germany	77
4	United States	72
5	Serbia	52
6	Hungary	51
7	France	46
8	Israel	39
9	Spain	36
⋮	⋮	⋮
22	India	26
23	Cuba	19
⋮	⋮	⋮
53	Mexico	5

Tabla 22: Grandmasters per country until July 2012.

B

UCI PROTOCOL

The **Universal Chess Interface** (UCI) [65] is a protocol that enables a chess engine to communicate with a graphical user interface (GUI).

The features of the UCI protocol are the following:

- This protocol is independent of the operating system.
- All communication is done with text commands via standard input and output.
- The chess engine must be able to process the standard input, even while thinking.
- All commands that the chess engine and GUI receive will end with "\n". Also, any number of blank spaces between tokens is allowed.
- The chess engine begins to calculate until it has received the current position through the command **position** followed by the command **go**.
- If the chess engine or the GUI receives an unknown token or command it should just ignore it.
- By default, the opening book handling is done by the GUI.
- The move format is in **long algebraic notation**. This notation describes the movement of a chess piece indicating its source square followed by its

target square, where each square is represented by its column followed by its row. For example, the move e2e4 denotes the move of a chess piece from the square e2 to square e4.

In Section B.1, we will describe the commands from the graphical user interface to the chess engine, and in Section B.2, we will describe the commands in the opposite direction.

B.1 FROM GUI TO CHESS ENGINE

The main commands from the graphical user interface to the chess engine are the following:

- **uci**. It tells to the chess engine to use the universal chess interface protocol. After the chess engine receives the uci command, it must identify itself with the id command and send the option commands to tell the GUI which settings the chess engine supports (if any). Finally, the chess engine should sent the uciok command to acknowledge the uci mode. If no uciok command is sent within a certain time period, the chess engine task will be killed by the GUI.

- **isready**. This command is used to synchronize the chess engine with the GUI. When the GUI has sent a command or multiple commands that can take some time to complete, this command can be used to wait for the engine to be ready again. This command must always be answered with readyok command when the chess engine is ready.

- **ucinewgame**. It is sent to the chess engine when the next search (started with position and go commands) corresponds to a new game. As the engine's reaction to ucinewgame command can take some time the GUI should always send the isready command after the ucinewgame command to wait for the chess engine to finish its operation.

- **position**. This command set up a position on the chess board. The syntax is the following:
  position [<fen> | startpos] <moves> ...
  where:
  <fen> denotes the position.
  startposis the command that denotes the start position.
  <fen> is the moves list.

- **go**. With this command the chess engine starts the search of the principal variation.
- **stop**. It stops the search of the principal variation as soon as possible.
- **quit**. It terminates the execution of the chess engine as soon as possible.

B.2 FROM CHESS ENGINE TO GUI

The main commands from the chess engine to the graphical user interface are the following:

- **id**. This command identifies the name and author of the chess engine. Its syntax is the following:
  id [name <chess engine name>| author <chess engine author>]
  where:
  <chess engine name> is the chess engine name.
  <chess engine author> is the chess engine author.
- **uciok**. This command must be sent after the id command to tell the GUI that the engine is ready in uci mode.
- **readyok**. This command must be sent when the chess engine has received an isready command and is ready to accept new commands.
- **bestmove**. The chess engine stops the search since it has found the best move in this position.

BIBLIOGRAPHY

[1] T. Bäck, J. M. de Graaf, J. N. Kok, and W. A. Kosters. Theory of genetic algorithms. In *Proceedings of the First European Conference on Artificial Life*, pages 263–271. MIT Press, 1997.

[2] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 14–21. Lawrence Erlbaum Associates (Hillsdale), 1987.

[3] D. Beal and M. C. Smith. Multiple probes of transposition tables. *ICCA Journal*, 19(4):227–233, 1996.

[4] L. B. Booker. Intelligent behavior as an adaptation to the task environment. Technical Report 243, University of Michigan, 1982.

[5] B. Bošković, J. Brest, A. Zamuda, S. Greiner, and V. Žumer. History Mechanism Supported Differential Evolution for Chess Evaluation Function Tuning. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 2011. (in press).

[6] B. Bošković, S. Greiner, J. Brest, and V. Žumer. A differential evolution for the tuning of a chess evaluation function. In *2006 IEEE Congress on Evolutionary Computation*, pages 1851–1856, Vancouver, BC, Canada, July 16-21 2006. IEEE Press.

[7] B. Bošković, S. Greiner, J. Brest, A. Zamuda, and V. Žumer. An Adaptive Differential Evolution Algorithm with Opposition-Based Mechanisms,

Applied to the Tuning of a Chess Program. In U. Chakraborty, editor, *Advances in Differential Evolution*, pages 287–298. Springer, Studies in Computational Intelligence, Vol. 143, Heidelberg, Germany, 2008.

[8] J. Branke. Evolutionary algorithms for neural network design and training. In J. Talander, editor, *Proceedings of the First Nordic Workshop on Genetic Algorithms and its Applications*, pages 145–163, Vaasa, Finland, 1995.

[9] A. Brindle. *Genetic Algorithms for Function Optimization*. PhD thesis, U. of Michigan, 1981.

[10] B. P. Buckles and F. E. Petry. *Genetic Algorithms*. 1992.

[11] M. S. Campbell and T. A. Marsland. A comparison of minimax tree search algorithms. *Artificial Intelligence*, 20(4):347–367, 1983.

[12] Y. Chauvin and D. E. Rumelhart, editors. *Backpropagation: theory, architectures, and applications*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1995.

[13] K. Chellapilla and D. Fogel. Evolution, neural networks, games, and intelligence. *Proceedings of the IEEE*, 87(9):1471 –1496, Sept. 1999.

[14] C. A. Coello Coello, A. D. Christiansen, and A. H. Aguirre. Using a new ga-based multiobjective optimization technique for the design of robot arms. *Robotica*, 16(4):401–414, July 1998.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, New York, 2001.

[16] C. Darwin. *The origin of species*. New York :P.F. Collier,. http://www.biodiversitylibrary.org/bibliography/24252.

[17] S. Das and P. N. Suganthan. Differential evolution: A survey of the state-of-the-art. *IEEE Trans. Evolutionary Computation*, pages 4–31, 2011.

[18] D. Dasgupta and D. R. Mcgregor. Designing application-specific neural networks using the structured genetic algorithm. In *In Proceedings of the International Conference on Combinations of Genetic Algorithms and Neural Networks*, pages 87–96. IEEE Computer Society Press, 1992.

[19] O. David-Tabibi, M. Koppel, and N. S. Netanyahu. Genetic algorithms for mentor-assisted evaluation function optimization. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, GECCO '08, pages 1469–1476, New York, NY, USA, 2008. ACM.

[20] O. David-Tabibi, M. Koppel, and N. S. Netanyahu. Expert-driven genetic algorithms for simulating evaluation functions. *Genetic Programming and Evolvable Machines*, 12:5–22, March 2011.

[21] O. David-Tabibi, H. J. van den Herik, M. Koppel, and N. S. Netanyahu. Simulating human grandmasters: evolution and coevolution of evaluation functions. In *GECCO'09*, pages 1483–1490, 2009.

[22] K. A. De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, Ann Arbor, MI, USA, 1975. AAI7609381.

[23] N. Dodd and R. Establishment. Optimisation of network structure using genetic techniques. In *International Symposium on Neural Networks*, 1990.

[24] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing (Natural Computing Series)*. Springer, Oct. 2008.

[25] T. Ellman. Explanation-based learning: a survey of programs and perspectives. *ACM Computing Surveys*, 21(2):163–221, June 1989.

[26] A. E. Elo. *The rating of chessplayers, past and present*. Arco Pub., New York, 1978.

[27] L. Fausett, editor. *Fundamentals of neural networks: architectures, algorithms, and applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[28] D. Fogel. An introduction to simulated evolutionary optimization. *IEEE Transactions on Neural Networks*, 5(1):3–14, January 1994.

[29] D. B. Fogel. *Evolutionary Computation: Towards a New Philosophy of Machine Intelligence*. Wiley-IEEE Press, 2nd edition, 1999.

[30] D. B. Fogel, T. J. Hays, S. L. Hahn, and J. Quon. A self-learning evolutionary chess program. *Proceedings of the IEEE*, 92(12):1947–1954, 2004.

[31] D. B. Fogel, T. J. Hays, S. L. Hahn, and J. Quon. Further evolution of a self-learning chess program. In *Proceedings of the 2005 IEEE Symposium on Computational Intelligence and Games (CIG05)*, pages 73–77, Essex, UK, April 4-6 2005. IEEE Press.

[32] D. B. Fogel, T. J. Hays, S. L. Hahn, and J. Quon. The blondie25 chess program competes against fritz 8.0 and a human chess master. In S. J. Louis and G. Kendall, editors, *Proceedings of the 2006 IEEE Symposium on Computational Intelligence and Games (CIG06)*, pages 230–235, Reno, Nevada, USA, May 22-24 2006. IEEE Press.

[33] L. J. Fogel. *Artificial Intelligence through Simulated Evolution*. John Wiley, New York, 1966.

[34] L. J. Fogel. *Intelligence through simulated evolution: forty years of evolutionary programming*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[35] F. Girosi and T. Poggio. Networks and the best approximation property. *Biological Cybernetics*, 63:169–176, 1989.

[36] M. E. Glickman. A comprehensive guide to chess ratings. *Amer. Chess Journal*, 3:59–102, 1995.

[37] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[38] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, 1991.

[39] D. E. Goldberg, K. Deb, and B. Korb. Don't worry, be messy. In *ICGA'91*, pages 24–30, 1991.

[40] D. Gomboc, M. Buro, and T. Marsland. Tuning evaluation functions by maximizing concordance. *Theoretical Computer Science*, 349(2):202–229, 2005.

[41] M. Gori and A. Tesi. On the problem of local minima in backpropagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14:76–86, 1992.

[42] M. Guid, M. Možina, J. Krivec, A. Sadikov, and I. Bratko. Learning positional features for annotating chess games: A case study. In *CG '08: Proceedings of the 6th international conference on Computers and Games*, pages 192–204. Springer. Lecture Notes in Computer Sciences, Vol. 5131, Heidelberg, Germany, 2008.

[43] M. H. Hassoun. *Fundamentals of Artificial Neural Networks*. MIT Press, Cambridge, MA, USA, 1st edition, 1995.

[44] A. Hauptman. Gp-endchess: Using genetic programming to evolve chess endgame players. In *In: Proceedings of 8th European Conference on Genetic Programming (EuroGP2005*, pages 120–131. Springer, 2005.

[45] A. Hauptman and M. Sipper. Evolution of an efficient search algorithm for the mate-in-n problem in chess. In *Proceedings of the 10th European conference on Genetic programming*, EuroGP'07, pages 78–89, Berlin, Heidelberg, 2007. Springer-Verlag.

[46] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Macmillan, New York, 1994.

[47] D. O. Hebb. *The organization of behavior: A neuropsychological theory*. Wiley, New York, June 1949.

[48] R. Hecht-Nielsen. *Neurocomputing / Robert Hecht-Nielsen*. Addison-Wesley Pub. Co., Reading, Mass., 1990.

[49] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992.

[50] R. Hooke and T. A. Jeeves. " direct search" solution of numerical and statistical problems. *J. ACM*, 8:212–229, April 1961.

[51] R. Hunter. Mm algorithms for generalized bradley-terry models. *The Annals of Statistics*, 32:2004, 2004.

[52] S. ichi Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5(4-5):185 – 196, 1993.

[53] J. S. Judd. Learning in neural networks. In *Proceedings of the first annual workshop on Computational learning theory*, COLT '88, pages 2–8, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

[54] G. Kendall and G. Whitwell. An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, volume 2, pages 995–1002. IEEE Press, May 2001.

[55] H. Kitano. Designing Neural Networks Using Genetic Algorithms with Graph Generation System. *Complex Systems Journal*, 4:461–476, 1990.

[56] B. Klein and D. Rossin. Data quality in neural network models: effect of error rate and magnitude of error on predictive accuracy. *Omega*, 27(5):569 – 582, 1999.

[57] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[58] T. Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69, 1982.

[59] T. Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464 –1480, sep 1990.

[60] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA, 1992.

[61] C.-T. Lin and C. S. G. Lee. *Neural fuzzy systems: a neuro-fuzzy synergism to intelligent systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[62] T. A. Marsland. *Computers, Chess, and Cognition*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1990.

[63] T. A. Marsland and M. Campbell. A survey of enhancements to the alpha-beta algorithm. In *Proceedings of the ACM '81 conference*, ACM '81, pages 109–114, New York, NY, USA, 1981. ACM.

[64] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5(4):115–133, Dec. 1943.

[65] S. Meyer-Kahlen. Uci protocol. http://www.shredderchess.com/chess-info/features/uci-universal-chess-interface.html, april 2009.

[66] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, second edition, 1996.

[67] M. L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[68] H. Nasreddine, H. Poh, and G. Kendall. Using an Evolutionary Algorithm for the Tuning of a Chess Evaluation Function Based on a Dynamic Boundary Strategy. In *Proceedings of 2006 IEEE international Conference on Cybernetics and Intelligent Systems (CIS'2006)*, pages 1–6. IEEE Press, 2006.

[69] M. Newborn. *Kasparov versus deep blue: computer chess comes of age*. Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[70] K. Price, R. M. Storn, and J. A. Lampinen. *Differential Evolution: A Practical Approach to Global Optimization (Natural Computing Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[71] F. Reinfeld. *One Thousand and One Winning Chess Sacrifices and Combinations*. Wilshire Book Company, 1969.

[72] H.-P. Schwefel. *Evolution and optimum seeking*. Sixth-generation computer technology series. Wiley, 1995.

[73] C. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 7(41):256–275, 1950.

[74] D. J. Slate and L. R. Atkin. *Chess Skill in Man and Machine*, chapter Chess 4.5 - The Northwestern University Chess Program, pages 82–118. Springer-Verlag, New York, 1977.

[75] R. S. Sutton and A. G. Barto. A temporal-difference model of classical conditioning. In *Ninth Annual Conference of the Cognitive Science Society*, pages 355–378, Hillsdale, New Jersey, USA, July 1987. Lawrence Erlbaum Associates, Inc.

[76] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press, Mar. 1998.

[77] K. Swingler. *Applying neural networks: A practical guide*. Academic Press, London, 1996.

[78] S. Thrun. Learning to play the game of chess. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems (NIPS) 7*, pages 1069–1076, Cambridge, MA, 1995. MIT Press.

[79] A. Turing. *Digital Computers Applied to Games, of Faster than Thought*, chapter 25, pages 286–310. Pitman, 1953.

[80] E. Vázquez-Fernández, C. Coello, and F. Sagols. Assessing the positional values of chess pieces by tuning neural networks' weights with an evolutionary algorithm. In *Proceedings of 2012 World Automation Congress (WAC 2012)*, Puerto Vallarta, México, 2012.

[81] E. Vázquez-Fernández, C. A. C. Coello, and F. D. S. Troncoso. Evolutionary algorithm with history mechanism for chess evaluation function. *Applied Soft Computing*.

[82] E. Vázquez-Fernández, C. A. C. Coello, and F. D. S. Troncoso. An evolutionary algorithm for tuning a chess evaluation function. In *2011 IEEE Congress on Evolutionary Computation*, New Orleans, Louisiana, USA, June 5–8 2011.

[83] E. Vázquez-Fernández, C. A. C. Coello, and F. D. S. Troncoso. An evolutionary algorithm coupled with the hooke-jeeves algorithm for tuning a chess evaluation function. In *IEEE Congress on Evolutionary Computation*, pages 1–8, Brisbane, Australia, 2012.

[84] A. Vellido, P. J. G. Lisboa, and J. Vaughan. Neural networks in business: a survey of applications (1992-1998). *Expert Systems with Applications*, 17(1):51–70, July 1999.

[85] D. Whitley. The genitor algorithm and selection pressure: why rank-based allocation of reproductive trials is best. In *Proceedings of the third international conference on Genetic algorithms*, pages 116–121, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[86] D. Whitley, S. Rana, and R. Heckendorn. Representation issues in neighborhood search and evolutionary algorithms, 1998.

[87] A. Zobrist. A new hashing method with application for game playing. Technical Report 88, The University of Wisconsin, Madison WI, USA, 1970. Reprinted (1990) in ICCA Journal, Vol. 13, No. 2, pp. 69-73.