School of Computer Science

Investigating Evolutionary Checkers by Incorporating Individual and Social Learning, N-tuple Systems and a Round Robin Tournament

by

Belal Al-Khateeb

BSc (Hons), MSc (Hons)

Thesis submitted to the University of Nottingham for
the degree of Doctor of Philosophy

June 2011

Abstract

In recent years, much research attention has been paid to evolving self-learning game players. Fogel's Blondie24 is just one demonstration of a real success in this field and it has inspired many other scientists. In this thesis, artificial neural networks are employed to evolve game playing strategies for the game of checkers by introducing a league structure into the learning phase of a system based on Blondie24. We believe that this helps eliminate some of the randomness in the evolution. The best player obtained is tested against an evolutionary checkers program based on Blondie24. The results obtained are promising. In addition, we introduce an individual and social learning mechanism into the learning phase of the evolutionary checkers system. The best player obtained is tested against an implementation of an evolutionary checkers program, and also against a player, which utilises a round robin tournament. The results are promising.

N-tuple systems are also investigated and are used as position value functions for the game of checkers. The architecture of the *n*-tuple is utilises temporal difference learning. The best player obtained is compared with an implementation of evolutionary checkers program based on Blondie24, and also against a Blondie24 inspired player, which utilises a round robin tournament. The results are promising. We also address the question of whether piece difference and the look-ahead depth are important factors in the Blondie24 architecture. Our experiments show that piece difference and the look-ahead depth have a significant effect on learning abilities.

List of Publications

During my PhD research programme, the following publications have been produced, representing the research work conducted for this thesis:

1- Al-Khateeb B. and Kendall G., Introducing a Round Robin Tournament into Blondie24, In Proceedings of the IEEE 2009 Symposium on Computational Intelligence and Games (CIG09), Milan, Italy, 2009, 112-116.

The work in chapter four is based on this paper.

2- Al-Khateeb B. and Kendall G., The Importance of a Piece Difference Feature to Blondie24**,** In Proceedings of the the 10th Annual Workshop on Computational Intelligence (UK2010), Essex, UK, 2010, 1-6.

The work in chapter seven is based on this paper.

3- Al-Khateeb B. and Kendall G., The Importance of look ahead Depth in Evolutionary Checkers, In Proceeding of the 2011 IEEE Congress on Evolutionary Computation (CEC 2011), New Orleans, USA, 2011.

The work in chapter seven is based on this paper.

4- Al-Khateeb B. and Kendall G., Introducing Individual and Social Learning into Evolutionary Checkers, Submitted (Second Review) to the Transactions on Computational Intelligence and AI in Games (TCIAIG), 2011.

The work in chapter five is based on this paper.

5- Al-Khateeb B., Kendall G. and Lucas S., Introducing *N*-Tuple Systems into Evolutionary Checkers, Submitted to the Transactions on Computational Intelligence and AI in Games (TCIAIG), 2011.

The work in chapter six is based on this paper.

Acknowledgements

It is a pleasure to thank those who made this thesis possible. First of all, I would like to thank my father Ismail Al-Khateeb and my mother Layla Salih for their unlimited support. They were always beside me when I needed incorporeal support. I would like to thank my wife Eman for her endless love and support.

Thanks are also due to Iraqi Government/Ministry of Higher Education and Scientific Research for the doctoral scholarship and all the financial support during my PhD study.

I would like to thank my supervisor, Professor Graham Kendall for sharing thoughts with me, for his time, encouragement, guidance, advice, and patience. This thesis would not have been possible without him.

Also I would like to thank the staff members of the Iraqi Cultural department/London for their continuous help and support.

Also, I would like to show my gratitude to my friends Ahmed Fuad and Ammar Albakaa for their help and support.

At last but not least, I would like to acknowledge the staff members of the School of Computer Science, ASAP Group and The International Office at the University of Nottingham.

Table of Contents

List of Tables

List of Figures

List of Algorithms

GLOSSARY

ANN	.....	Artificial Neural Network.
Blondie24	.....	An evolutionary checkers program, produced by David Fogel.
Blondie24-RR	.....	A round robin player, based on the implementation of Blondie24.
Blondie24-RR1Ply	.....	A round robin player, based on the implementation of Blondie24. This player trained using one ply depth.
Blondie24-RR2Ply	.....	A round robin player, based on the implementation of Blondie24. This player trained using two ply depth.
Blondie24-RR3Ply	.....	A round robin player, based on the implementation of Blondie24. This player trained using three ply depth.
Blondie24-RRNPD	.....	A round robin player, based on the implementation of Blondie24 without the piece difference feature.
C_0	.....	Base line player, based on the implementation of Blondie24.
C_0-NPD	.....	Base line player, based on the implementation of Blondie24 without the piece difference feature.
C_1	.....	An individual and social learning player, with only one player saved in the social pool.
C_{10}	.....	An individual and social learning player, with the values of 5 and 10 for deciding where the individual and social phases occur.
C_{100}	.....	An individual and social learning player, with the values of 50 and 100 for deciding where the individual and social phases occur.
C_{10}-1Ply	.....	An individual and social learning player, with the values of 5 and 10 for deciding where the individual and social phases occur. This player trained using one ply depth.
C_{10}-4Ply	.....	An individual and social learning player, with the values of 5 and 10 for deciding where the individual and social phases occur. This player trained using four ply depth.
C_{10}-NPD	.....	An individual and social learning player, with the values of 5 and 10 for deciding where the individual and social phases occur. This player is constructed with the piece difference feature.
C_{10}-RR	.....	A round robin individual and social learning player, with the values of 5 and 10 for deciding where the individual and social phases occur.

C_1-N0.0001	.....	1-tuple with alpha value=0.0001 temporal and difference learning checkers player.
C_1-N0.001	.....	1-tuple with alpha value=0.001 temporal and difference learning checkers player.
C_1-N0.01	.....	1-tuple with alpha value=0.01 temporal and difference learning checkers player.
C1Ply	.....	An evolutionary checkers player, based on the implementation of Blondie24. This player trained using one ply depth.
C_1-tuple	.....	1-tuple evolutionary checkers player.
C_{20}	.....	An individual and social learning player, with the values of 10 and 20 for deciding where the individual and social phases occur.
C_{200}	.....	An individual and social learning player, with the values of 100 and 200 for deciding where the individual and social phases occur.
C2Ply	.....	An evolutionary checkers player, based on the implementation of Blondie24. This player trained using two ply depth.
C3Ply	.....	An evolutionary checkers player, based on the implementation of Blondie24. This player trained using three ply depth.
C4Ply	.....	An evolutionary checkers player, based on the implementation of Blondie24. This player trained using four ply depth.
C_{50}	.....	An individual and social learning player, with the values of 20 and 50 for deciding where the individual and social phases occur.
C_5-N0.0001	.....	5-tuple with random walk and alpha value=0.0001 temporal and difference learning checkers player.
C_5-N0.001	.....	5-tuple with random walk and alpha value=0.001 temporal and difference learning checkers player.
C_5-N0.001-1Ply	.....	5-tuple with random walk and alpha value=0.001 temporal and difference learning checkers player. This player trained using one ply depth.
C_5-N0.001-4Ply	.....	5-tuple with random walk and alpha value=0.001 temporal and difference learning checkers player. This player trained using four ply depth.
C_5-N0.001-NPD	.....	5-tuple with random walk and alpha value=0.001 temporal and difference learning checkers player. This player is constructed with the piece difference feature.
C_5-N0.01	.....	5-tuple with random walk and alpha value=0.01 temporal and difference learning checkers player.
C_5-tuple	.....	5-tuple with random walk evolutionary checkers player.
CIGAR	.....	Case-Injected Genetic Algorithm.
EA	.....	Evolutionary Algorithm.

EANN	.....	Evolutionary Artificial Neural Network.
EP	.....	Evolutionary Programming.
ES	.....	Evolution Strategies.
GA	.....	Genetic Algorithm.
GP	.....	Genetic Programming.
NEAT	.....	NeuroEvolution of Augmenting Topologies.
Piece Difference	.....	The difference of the number of the player pieces currently on the board and the number of the opponent pieces currently on the board.
PLY	.....	The ply of a node is the number of moves needed to reach that node (i.e. arcs from the root of the tree). The ply of a tree is the maximum of the plies of its nodes.
SANE	.....	Symbiotic Adaptive Neuro-Evolution.
TDL	.....	Temporal Difference Learning.

Chapter One

Introduction

1.1 INTRODUCTION

The motivation for the work carried out in this thesis is inspired from Fogel's success in checkers in which his program, Blondie24 (Chellapilla and Fogel 2001; Fogel and Chellapilla 2002) was able to play a game of checkers at the human expert level, injecting as little expert knowledge as possible into the algorithm. Fogel combined evolution strategies with neural networks and used a minimax search tree as a look-ahead mechanism to find potentially good moves for the game of checkers. Blondie24 only received feedback of its performance after a certain number of games, not knowing the result of individual games.

Blondie24 represents a landmark in evolutionary learning. Even so, it has still attracted comments about its design. One of them is concerned with the piece difference feature and how it affects the learning process of Blondie24. Although, there has been a lot of discussion about the importance of the look-ahead depth level used in Fogel's work. In this thesis we also address the question of whether piece difference is an important factor in the Blondie24 architecture. Although this issue has been addressed before, this work provides a different experimental setup to previous work, but arrives at

the same conclusion. Our experiments show that piece difference has a significant effect on learning abilities. Finally a detailed investigation of the importance of the look-ahead depth is carried out. We believe this is the first time such an intensive study has been done for evolutionary checkers. Our experiments show that increasing the depth of a look-ahead has significant improvements on the performance of the checkers program and has a significant effect on its learning abilities.

One other thing that can be noticed from the design of Blondie24 is that the strategies do not all play the same number of games because, by chance, some would be selected as opponents more often than others. Our research will investigate if this is a limiting factor in order to eliminate the randomness in choosing opponents. Thirty feed forward neural network players are played against each other, using a round robin tournament structure, for 140 generations and the best player obtained is tested against an evolutionary checkers program based on Blondie24. We also test the best player against an online program, as well as two other strong programs. The results obtained are promising.

The work in this thesis is also inspired from the success of Su and Kendall's work (Kendall and Su 2003 and Su 2005). Su investigated imperfect evolutionary systems in her PhD thesis, using the stock market as a problem domain (Su 2005). In (Kendall and Su 2003), an investigation of the integration of individual and social learning of multi-agent based models in a simulated stock market was carried out, where the evolved neural network traders learn to trade their stocks, giving the investors' higher

returns compared to a baseline buy-and-hold strategy. So we decided to introduce an individual and social learning mechanism into the learning phase of the evolutionary checkers system. The best player obtained is tested against an implementation of an evolutionary checkers program, and also against a player, which utilises a round robin tournament. The results are promising and demonstrate that using individual and social learning enhances the learning process of the evolutionary checkers system and produces a superior player compared to what was previously possible. In addition, we conduct an investigation to choose which values should be used when deciding where the individual and social learning phases should occur.

The success of *n*-tuple systems in many applications including optical character recognition, and evolving game playing strategies for the game of Othello (Lucas 2008) provides the inspiration to also apply the *n*-tuple systems in this thesis. *N*-tuple systems are investigated and are used as position value functions for the game of checkers. The architecture of the *n*-tuple is utilises temporal difference learning. The best player obtained is compared with an implementation of evolutionary checkers program based on Blondie24, and also against a Blondie24 inspired player, which utilises a round robin tournament. The results are promising and demonstrate that using *n*-tuple enhances the learning process of checkers and produces a good player. The conclusion is that *n*-tuple systems learn faster when compared against other approaches. In addition, an investigation of learning rates for temporal difference learning is carried out.

This chapter is structured as follows: Section 1.2 describes the contributions of this thesis, while section 1.3 outlines the structure. A summary of the chapter is presented in section 1.4.

1.2 CONTRIBUTIONS

This thesis makes the following contributions:

1- Introducing a round robin tournament into the evolutionary phase of the evolutionary checkers program, aiming to eliminate the randomness and hence produce a better player. This work is presented in chapter four.

2- Introducing individual and social learning into an evolutionary checkers in order to enhance its learning ability and hence produce a superior player. In addition, we show that individual and social learning has a wider applications area, in addition to the stock market. This work is presented in chapter five.

3- Investigating the use of a round robin tournament within the individual and social learning framework, aiming to eliminate the randomness, and producing a superior player. This work is presented in chapter five.

4- Introducing *n*-tuple systems into evolutionary checkers, producing a good player, whilst using less computational time than is required for the evolutionary checkers player in step 1. This work is presented in chapter six.

5- Investigating the importance of piece difference in evolutionary checkers by showing the effects of using/not using it. This work is presented in chapter seven.

6- Investigating the importance of the look-ahead depth in evolutionary checkers by showing the effects of using/not using it. This work is presented in chapter seven.

1.3 THESIS OUTLINE

This thesis is structured as follows; Chapter two presents the background of this thesis. It starts with basic algorithms such as minimax tree search together with alpha-beta pruning. A literature review in evolutionary computation, artificial neural networks and evolutionary neural networks is also presented in this chapter. The chapter continues with a discussion on various computer game programs, focussing on those that have employed evolutionary methodologies. The chapter also describes Fogels' Blondie24 checkers program. The review of Su's work on individual and social learning, together with a review of *n*-tuple systems, are presented before concluding the chapter with a summary.

Chapter three presents various preliminaries for the evolutionary checkers that will be used throughout this thesis. These preliminaries include the implementation of an evolutionary checkers program named C_0, which is based on the Blondie24 architecture. It also includes the description of the two-move ballot and the standard rating formula as a way to test the

outcome of applying the proposed methods that will be used to enhance the learning process of C_0.

Chapter four presents a round robin tournament as a proposed method to eliminate the randomness in the evolutionary phase of C_0 in order to enhance its learning ability and produce a better player. The resultant player, named Blondie24-RR, is tested against C_0 using the idea of two-move ballot and the standard rating formula. Blondie24-RR also tested against an online program and two strong programs.

Chapter five introduces individual and social learning to the evolutionary checkers algorithms. Many experiments are carried out in order to determine the best values that can be used to decide where the individual and social phases should occur. The player with the best values, named C_{10}, plays against C_0 and Blondie24-RR, using the two-move ballot and standard rating formula to test the outcome. Also we decided to use round robin tournament with the individual and social learning, and the resultant player named C_{10}-RR plays against C_0, Blondie24-RR and C_{10} using the two-move ballot and standard rating formula.

Chapter six introduces *n*-tuple systems into two evolutionary checkers programs, one based on C_0 and the other using temporal difference learning. Various experiments are carried out to determine the best settings for the *n*-tuple. All the resultant players are set to play against C_0, Blondie24-RR, C_{10} and C_{10}-RR using two-move ballot and standard rating formula to test the outcome.

The final experiments in this thesis are presented in chapter seven, where we show the importance of the piece difference feature and the look-ahead depth to all evolutionary checkers programs, constructed using the proposed methods in chapters three, four, five and six.

We summarise the contributions of this thesis in chapter eight, together with suggestions as how the work and ideas presented in this thesis could be further developed.

1.4 SUMMARY

This chapter has presented the works that have inspired this thesis. It also described the main contributions of the work presented in this thesis. Finally the thesis structure is presented. The next chapter will present the literature review for various artificial intelligence methods, some of which will be used in the development of this thesis.

Chapter Two

Literature Review

2.1 INTRODUCTION

This chapter discusses the research that has been conducted with respect automated game playing. Computer board games that have been associated with artificial intelligence techniques will also be discussed along with a discussion of evolutionary computation, individual and social learning and *n*-tuple systems, as learning techniques that have been utilised in automated game playing.

This chapter has been structured as follows; Section 2.2 describes the basic algorithms that are used by automated computer games. Evolutionary computation will be described in section 2.3. In section 2.4 a description of artificial neural networks is presented. Section 2.5 showed the various computer games programs. Blondie24 is presented in Section 2.6. Sections 2.7, 2.8 and 2.9 described individual and social learning, *n*-tuple systems and temporal difference learning respectively. Finally a summary for this chapter is presented in section 2.10.

2.2 BASIC ALGORITHMS

In general, for one person games and puzzles, a simple A* algorithm (Hart et. al. 1968) can be used to find the best move (Rich and Knight 1991; Nilsson 1998; Carter 2007). The A* algorithm is not suitable for complex two person games and a minimax search algorithm is commonly used to find the best move in these types of games (Kaindl 1990; Nilsson 1998; Carter 2007; Luger 2008). A minimax algorithm, in its general form, performs a complete depth first search by producing the whole game tree and then, using an evaluation function (which could represent the exact result such as win, lose or draw if the full game tree can be produced, or a heuristic value if the full search tree cannot be built), it computes the value of each leaf node. The algorithm then selects the best values and propagates these up towards the root of the tree. A best next move is selected to maximise the evaluation function. Algorithm 2.1 shows a typical minimax algorithm[1].

- Two players take turns and try respectively to maximize and minimize a scoring function.
- The two players are called respectively MAX and MIN.
- The MAX player makes the first move.
- Players take turns; successive nodes represent positions where different players must move.
- MAX node means the MAX player must move at that node.
- MIN nodes means MIN player must move at that node.
- The leaves represent terminal positions, i.e. positions where MAX wins or MIN wins.

```
function MINIMAX(N) is
        begin
          if N is a leaf then
             return the estimated score of this leaf
          else
             Let N1, N2, .., Nm be the successors of N;
             if N is a Min node then
                      return min{MINIMAX(N1), .., MINIMAX(Nm)}
             else
                      return max{MINIMAX(N1), .., MINIMAX(Nm)}
        end MINIMAX;
```

Algorithm 2.1 Minimax algorithm.

[1] http://www. cis.temple.edu/~ingargio/cis587/readings/alpha-beta.html.

Although minimax is able to find the best move in complex games, it is time consuming especially for larger search spaces. For example chess has an average branching factor of 35 (Russell and Norvig 2010) and a complete game tree could have about 35^{100} different positions to evaluate and search, which is impractical using methodologies which require a full search tree to be built. In order to address this problem, by being able to prune the search tree, alpha-beta (or α.β) search was introduced, which seeks to reduce the number of nodes that are evaluated in the search tree when using the minimax algorithm (Hsu1990; Rich and Knight 1991; Norvig 1992; Junghanns 1998; Luger 2008; Russell and Norvig 2010). α.β search is commonly used for two-player games. It stops evaluating moves, in a particular part of the search tree, when at least one possibility has been found that proves the move to be worse than a previously examined move. These moves, and more importantly those lower in the search tree, need not be evaluated further.

Alpha-beta pruning returns exactly the same result as minimax but can drastically reduce the size of the search space. Algorithm 2.2 presents a typical alpha-beta pruning algorithm[2]. Figure 2.1 shows how the alpha beta pruning works.

- Two players take turns and try respectively to maximize and minimize a scoring function.
- The two players are called respectively MAX and MIN.
- The MAX player makes the first move.
- Players take turns; successive nodes represent positions where different players must move.
- MAX node means the MAX player must move at that node.
- MIN nodes means MIN player must move at that node.
- The leaves represent terminal positions, i.e. positions where MAX wins or MIN wins.
- ALPHA value of a node is a value never greater than the true score of this node. Initially it is the score of that node, if the node is a leaf, otherwise it is -infinity. Then at a MAX node it is set to the largest of the scores of its successors explored up to now, and at a MIN node to the alpha value of its predecessor.

[2] http://www. cis.temple.edu/~ingargio/cis587/readings/alpha-beta.htm.

- BETA value of a node is a value never smaller than the true score of this node. Initially it is the score of that node, if the node is a leaf, otherwise it is +infinity. Then at a MIN node it is set to the smallest of the scores of its successors explored up to now, and at a MAX node to the beta value of its predecessor.

```
function Alpha-Beta(N, A, B) is ;; Here A is always less than B
        begin
          if N is a leaf then
             return the estimated score of this leaf
          else
                    Set Alpha value of N to -infinity and
                    Beta value of N to +infinity;
             if N is a Min node then
                    For each successor Ni of N loop
                         Let Val be Alpha-Beta (Ni, A, Min{B,Beta of N});
                         Set Beta value of N to Min{Beta value of N, Val};
                         When A >= Beta value of N then
                                  Return Beta value of N endloop;
                    Return Beta value of N;
             Else
                  For each successor Ni of N loop
                       Let Val be Alpha-Beta (Ni, Max{A,Alpha value of N}, B);
                       Set Alpha value of N to Max{Alpha value of N, Val};
                       When Alpha value of N >= B then
                            Return Alpha value of N endloop;
                  Return Alpha value of N;
        end Alpha-Beta;
```

Algorithm 2.2 Alpha-Beta pruning algorithm.

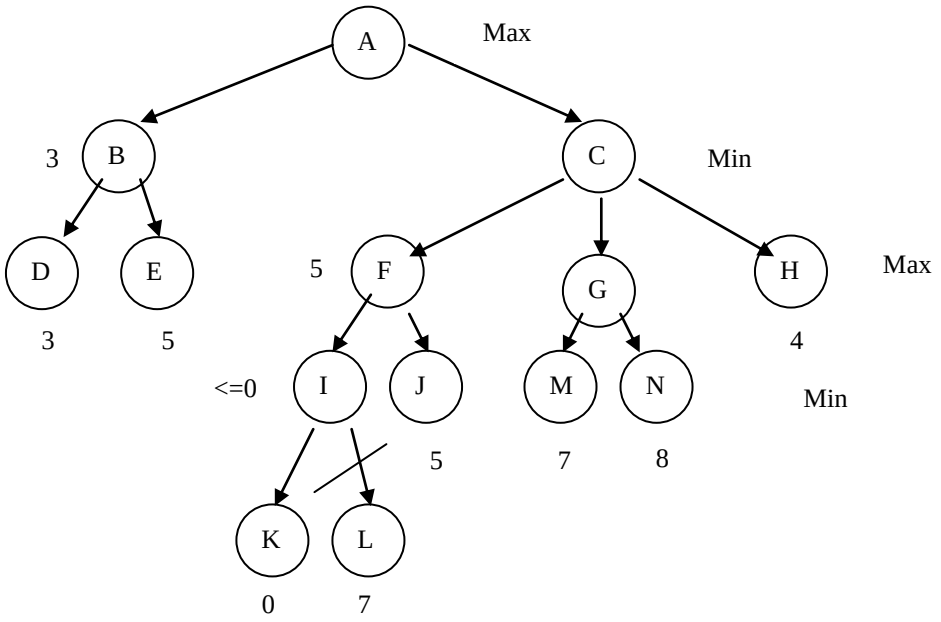

Figure 2.1 Example of Alpha-Beta pruning.

In the figure above, the entire tree headed by *B* is searched and hence can expect a score of at least 3. At *A*, when this alpha value is passed to *F*, it will enable us to skip the exploration of *L*. This is because after *K* is examined, *I* is guaranteed a maximum score of 0 (i.e *F* is guaranteed a minimum score of 0). But this is less than alphas value of 3, so no more branches of *I* need to be considered on examining *J*, *F* is assigned a value of 5. This value becomes the value of beta at node *C*.

2.3 EVOLUTIONARY COMPUTATION

Yao (1999a) defines evolutionary computation as the study of computational systems that use ideas and inspirations from natural evolution and adaptation. Although there is no strict definition about the different kinds of evolutionary computation, this section describes three variants: Genetic Algorithms (GA), Genetic Programming (GP) and Evolutionary Algorithms (EA), which can be further sub-divided into evolution strategies and evolutionary programming.

2.3.1 Evolutionary Algorithms

Evolutionary algorithms (Fogel 1994, 2006) are another form of evolutionary computation. Evolutionary algorithms focus on the potential solution of the problem, in contrast to genetic algorithms, which focus on the encoding structure of the problem (Fogel 1994). The structures that are used in evolutionary algorithms are problem dependant, which introduce a more natural representation than the general representation (often bit strings, at

least in early works) used in genetic algorithms. Another differentiating factor between evolutionary and genetic algorithms is that the former emphasizes the behavioral link between parents and offspring while the latter focuses on the genetic link (Fogel 2006). Evolutionary algorithms can be extended to Evolution Strategies (ES) and Evolutionary Programming (EP). The major difference between them is in the representation of the problem and the reproduction operators employed, where ES has a matrix of mutation vectors that corresponds to the population of chromosomes in which each gene in each chromosome has its own mutation standard deviation that evolves along with the chromosome, therefore the algorithm has self-adaptive mutation. An EP has one mutation value per chromosome, or one for the entire population. Evolution strategies were developed as a methodology for dealing with problems of numerical optimisation (Rechenberg 1965; Schwefel (1965, 1981)), where vectors of real numbers, instead of binary strings, were used to represent potential solutions. The distinctive characteristics of evolution strategies, in general, are Gaussian mutation, and discrete or intermediate recombination. Below are the two main schemes of deterministic selection in evolution strategies (Yao 1999b):

- (μ+λ): μ parents are used to create λ offspring. All individuals, (i.e. the (μ+λ) solutions) compete and the best μ solutions are selected as parents for next generation.

- (μ,λ): μ parents are used to create λ offspring, but only the λ offspring compete for survival and the μ parents are completely replaced each generation. Algorithm 2.3 shows an implementation

of (μ,λ) evolution strategies.

1- Generate the initial population of μ individuals. Each individual is a real-valued n-dimensional vector, where n is the number of parameters to be optimized.

2- Evaluate the fitness value for each individual of the population.

3- Generate λ offspring by adding a Gaussian random variable with zero mean and preselected standard deviation to each dimension of an individual.

4- Evaluate the fitness of each offspring.

5- Sort the λ offspring into a non-descending order according to their fitness values, and select the μ best offspring out of λ to be parents of the next generation.

6- Stop if the stopping criterion is satisfied; otherwise go to step 3.

Algorithm 2.3 An implementation of (μ,λ) evolution strategies (Yao 1999b).

Algorithm 2.3 describes mutation-based evolution strategies, i.e. offspring are generated by applying Gaussian mutations to parents. The Gaussian mutation operation used in Fogel (2006) is described by the following:

$$s_i' = s_i \cdot \exp(t' \cdot N(0,1) + t \cdot N_i(0,1)). \quad (2.1)$$

$$x_i' = x_i + N(0, s_i'). \quad (2.2)$$

Where $N(0,1)$ represents a single standard Gaussian random variable, $N_i(0,1)$ represents the i^{th} independent identically distributed standard Gaussian, and t and t' are operator-set parameters affecting global and individual step sizes. Evolution strategies make use of recombination operators for the process of producing new offspring. Discrete recombination and intermediate recombination are the two main recombination operators that are most frequently employed. Discrete recombination resembles uniform crossover in GAs where new offspring are generated by arbitrarily mixing components from the parents. In the intermediate recombination the vectors of two

parents are averaged together, element by element, to form a new offspring as shown in figure 2.2.

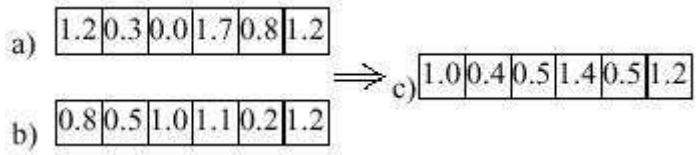

Figure 2.2 Example of intermediate recombination.

Evolutionary Programming employs vectors of real numbers as its representation of potential solutions for problem solving (Fogel et al. 1966; Bäck and Schwefel 1993; Fogel 1994). The absence of recombination and crossover in evolutionary programming is what most notably distinguishes it from evolution strategies. Instead, it employs some sort of tournament selection as a selection scheme and Gaussian mutation as the only reproduction operator. Algorithm 2.4 illustrates a typical application of evolutionary programming algorithms.

1- Generate the initial population of μ individuals.

2- Evaluate the fitness value for each individual of the population.

3- Each parent creates a single offspring by means of Gaussian mutation.

4- Evaluate the fitness of each offspring.

5- Conduct pairwise comparison over the union of parents and offspring. For each individual, q opponents are chosen uniformly at random from all the parents and offspring. For each comparison, if the individual's fitness is no smaller than the opponent's, it receives a "win."

6- Select μ individuals from the union of both the parents and the offspring (generated by Gaussian mutation) that have the most wins to be parents of the next generation.

7- Stop if the stopping criterion is satisfied; otherwise go to step 3.

Algorithm 2.4 An implementation of evolutionary programming algorithms (Bäck and Schwefel 1993; Yao 1999b; Fogel 2000).

Algorithm 2.4 is very similar to algorithm 2.3 as the only difference between them is in the reproduction operators.

For the optimisation of real-valued numerical parameters, evolution strategies and evolutionary programming, with real-value representations and Gaussian mutation, have been shown to have practical uses (Michalewicz 1992).

2.4 ARTIFICIAL NEURAL NETWORKS

Artificial neural networks are based on the idea of natural systems in which a set of neurons conduct transmission and communication processes travelling through axon (a long, slender projection of a nerve cell, or neuron, that conducts electrical impulses away from the neuron's cell body) connections (Patterson 1996; Coppin 2004; Galushkin 2007). An axon is one of two types of protoplasmic protrusions that extrude from the cell body of a neuron, the other type being dendrites(are the branched projections of a neuron that act to conduct the electrochemical stimulation received from other neural cells to the cell body). Axons are distinguished from dendrites by several features, including shape, length and function. Axons make contact with other cells at junctions called synapses. At a synapse, the membrane of the axon closely adjoins the membrane of the target cell, and special molecular structures serve to transmit electrical or electrochemical signals across the gap. The neurons constitute points that are able to adjust to new conditions, going through a process of learning from examples, and retaining that knowledge for future use (Pandya and Macy 1996). This section

discusses artificial neural networks in the context of network architecture, learning approaches, and also focuses on evolutionary artificial neural networks.

2.4.1 Perceptrons and Multi-layer Perceptrons

One of the first models introduced to categorize patterns through the process of observed learning is the perceptron (Rosenblatt 1959). Figure 2.3 shows a perceptron. A set of inputs represented as $x_1,x_2,....,x_m$ is received by the processing unit. A special input, b_k, termed a bias, which has its own weight (either fixed to +1 or variable). There is an associated weight (w_{kj}), which represents the connection between the processing unit *k* and an input x_i. A non-linear activation function, represented as φ(.), transforms the summed input to produce the output from the perceptron.

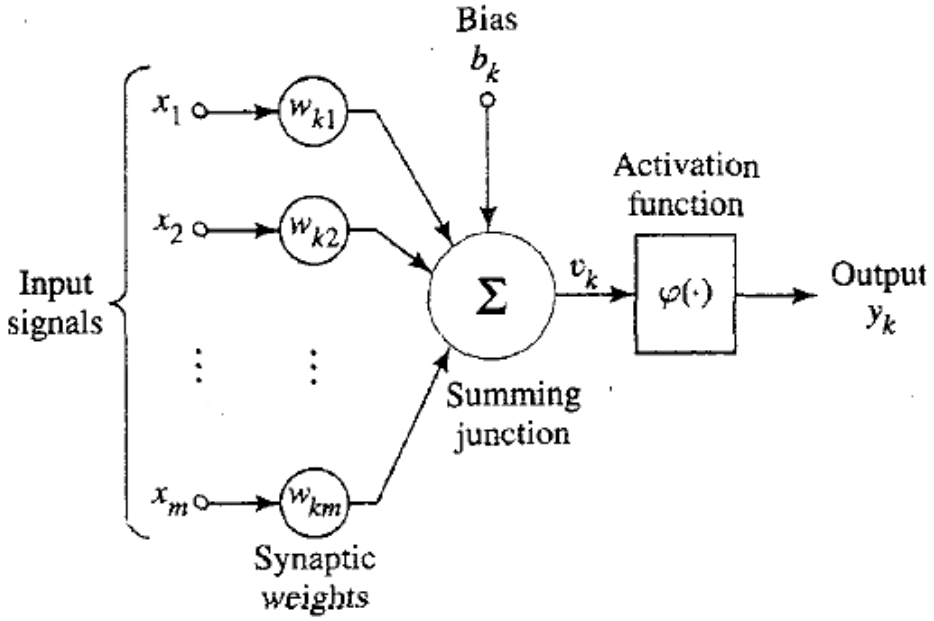

Figure 2.3 Single Perceptron (Haykin 1999).

The common activation functions are shown in table 2.1.

Name	Formula	Range of Output
Step function	Step(x) = 1 if $x \geq 0$, else 0.	0 or 1
Sign function	Sign(x) = +1 if $x \geq 0$, else -1	± 1
Sigmoid function	Sigmoid(x)=$1/(1+e^{-x})$	(0,1)
Hyperbolic function	Tanh(x)=$(e^{x}-e^{-x}) / (e^{x}+e^{-x})$	(-1,1)

Table 2.1 Some commonly used non-linear activation functions in artificial neural networks.

Several perceptrons can be grouped together to form a neural network where two layers of neurons are fully interconnected, but there is no interconnection between neurons in the same layer. Figure 2.4 shows a two layer perceptron network.

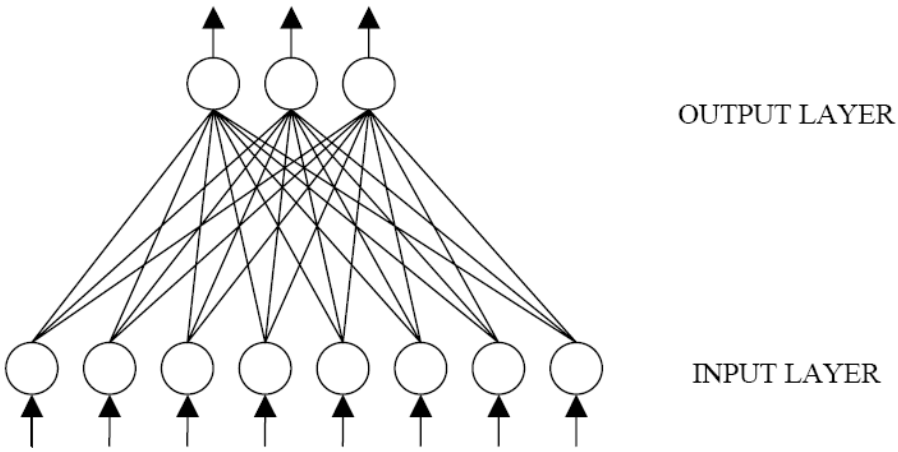

Figure 2.4 A two layer perceptron (Haykin 1999).

A learning task for a perceptron is to tune its weights using the optimiser in order to make the network produce the desired output for given inputs. There are many learning rules that can be applied to learn the network, please refer to Haykin (1999) for full details.

Two-layer perceptrons can be successfully trained for solving a number of function approximation and pattern classification problems, for which Rosenblatt (1962) shows the convergence properties of the perceptron learning rule. With regards to the two-layered perceptron, Minsky and Papert (1969), in their milestone book, proved that it has limited representational capabilities in representing non-linearly separable functions, even if they were as simple as XOR. Linearly separable means that a pattern can be separated into two classes by a single line (or a plane in higher dimensions). The architecture of a multi-layer perceptron with two hidden layers and three outputs are shown in Figure 2.5. Signals only pass in a forward direction (left to right in figure 2.5). Networks usually utilise one of the activation functions shown in Table 2.1.

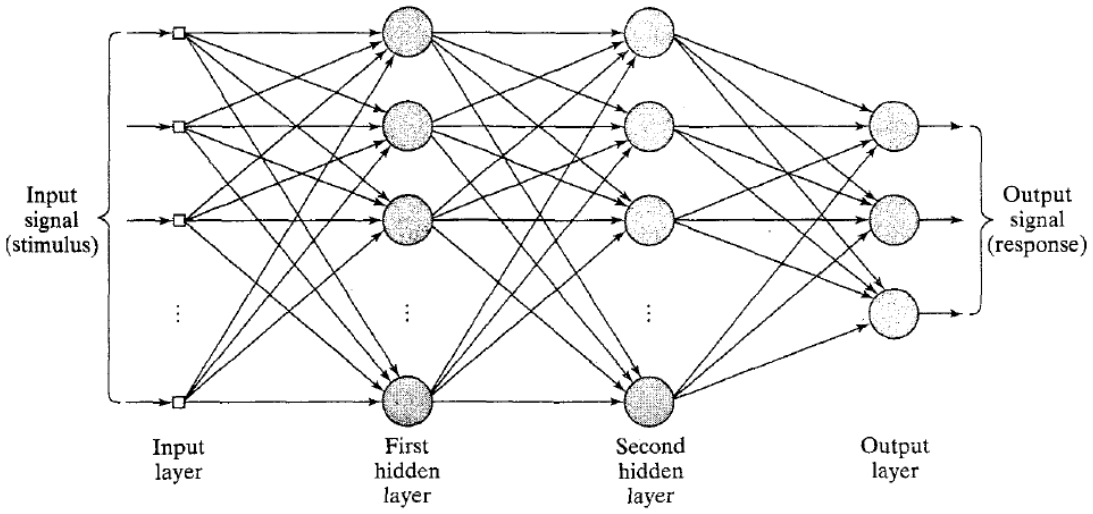

Figure 2.5 A multi-layer perceptron (Haykin 1999).

2.4.2 Backpropagation Learning and Other Neural Networks Models

Paker (1985) and Rumelhart et al. (1986) introduced the backpropagation algorithm for the training of multi-layer networks. The backpropagation

algorithm offers an efficient computational method for training multilayer networks, and overcome the problems highlighted by Minsky and Papert (1969). The objective is to train the network weights so as to minimise the Least-Square-Error (Zurada 1996) between the desired and the actual output. Algorithm 2.5 presents a backpropagation algorithm for learning a multi-layer feedforward network with one hidden layer:

```
Initialise the weights in the network (often randomly)
  Do
      For each example e in the training set
                  - O = neural-net-output(network, e) ; forward pass
                  - T = teacher output for e
                  - Calculate error (T - O) at the output units
                  - Compute delta_wh for all weights from hidden layer to  output layer ;
                    backward pass
                  - Compute delta_wi for all weights from input layer to hidden layer ; backward
                    pass continued
                  - Update the weights in the network
  Until all examples classified correctly or stopping criterion satisfied.
  Return the network.
```

Algorithm 2.5 Backpropagation algorithm for one hidden layer (Werbos 1994).

The backpropagation method is essentially a gradient descent method that minimises the error between the target output and the actual output from the network. More on the mathematical analysis of the backpropagation algorithm and delta rules may be obtained in Fausett (1994), Patterson (1996) and Russell and Norvig (2010). Other neural network topologies have also been proposed. Neural networks with one or more feedback loops are categorized as recurrent networks. The feedback may be of local or global type. Figure 2.6 demonstrates a basic recurrent network.

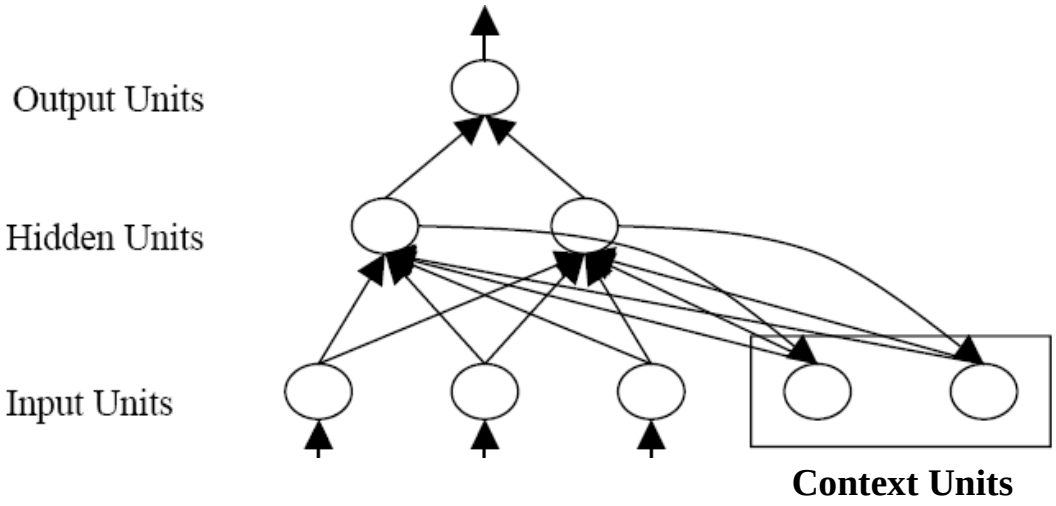

Figure 2.6 A simple recurrent neural network (Elman 1990).

Gradient-based is one of the many suggested training methods for training recurrent networks, as proposed in Williams and Zipser (1995). It is worth to mention that training recurrent networks is not as easy task, as the feedback networks mean the feed-forward training patterns are not fully known. Recurrent neural networks, in which all connections are symmetric, are referred to as Hopfield networks (Hopfield 1982), i.e., there must be a connection from unit *j* feedback to unit *i* whenever there is a connection from unit *i* to unit *j*. Among other commonly employed neural network models are Radial basis function networks (Park and Sandberg 1991), probabilistic neural networks (Specht 1990), and Kohonen self-organizing maps (Kohonen 1997). Readers are referred to Anderson and Rosenfield (1988), Fausett (1994) and Callan (1999) for further information about these network types.

It has been found that Multi-layer perceptrons with backpropagation learning are effective and efficient in solving a number of practical problems. For example financial time series predictions (Zirilli 1996), computer game

playing (Tesauro and Sejnowski 1989), and industrial applications such as steel scheduling (Schlanget al. 1996). However, there are some drawbacks with backpropagation training. One of these is the minimisation of the mean square error over all training examples. Another drawback is that it is necessary to calculate a derivative, which is computationally expensive. The learning is also liable to get trapped at a local minimum (Sutton 1986). A further consideration in using backpropagation is that it is still an *art* rather that a *science* to derive the network architecture. ANN research has not yet precisely identified any protocols to follow in terms of the number of layers and hidden units and the type of activation functions that should be used. In most cases, the design of a network will be subject to experience or repetitive tests using a different number of layers and hidden units. Evolutionary approaches for learning artificial neural networks have been explored for the purpose of tackling such problems.

2.4.3 Evolutionary Artificial Neural Networks

Yao (1999a) has described the introduction of an evolutionary learning approach into artificial neural networks at three different levels; namely, connection weights, network architectures, and learning rules. Connection weights could be evolved by utilising methodologies such as genetic algorithms. These algorithms offer a global search method for training the weights of the network and could help the problems of becoming being trapped in a local minima caused by gradient descent learning. Without human intervention, both the weights and the structure of artificial neural networks could be evolved automatically by evolving network architectures

using an evolutionary approach. The evolution of learning rules can be considered as a process of "learning how to learn" in artificial neural networks, where the adaptation of learning rules is attained through evolution. Please refer to Moriarty and Miikkulainen (1997), Yao (1999a), Miikkulainen (2007) and Yao and Islam (2008) for comprehensive surveys on evolutionary artificial neural networks.

2.4.3.1 Evolving Connection Weights

In artificial neural networks, there are two major phases in the training of the weights. The first phase is to decide on the representation of the connection weights, which is typically either binary strings or real-valued vectors; while the second phase is to decide the genetic operators to be used for the evolutionary process, in conjunction with the representation scheme. A typical evolutionary algorithm is illustrated in algorithm 2.6.

1- Decode each individual (a chromosome represents all connection weights) in the current generation into a set of connection weights and construct a corresponding ANN with the weights.

2- Evaluate each ANN by computing its total mean square error between actual and target outputs. Other error functions can also be used and problem-dependent. The higher the error, the lower the fitness. A regularization term may be included in the fitness function to penalize large weights.

3- Select parents with higher fitness for reproduction.

4- Apply search operators, such as crossover and/or mutation, to parents to generate offspring, which form the next generation of potential connection weights.

Algorithm 2.6 A typical algorithm for evolving connection weights in evolutionary artificial neural networks(Yao 1999a).

Real numbers are usually used to represent connection weights. However, early works in evolutionary artificial neural networks (Caudell and Dolan 1989; Garis 1991) showed that binary strings can also be exploited to

represent the connection weights. In a binary representation, each connection weight is represented by a number of bits with a certain length. Moreover, in a binary representation, several encoding methods such as the uniform method or the exponential method can be employed to encode real valued weights into binary bits using various ranges and precisions. An important issue for binary representation is the tradeoff between the precision of binary representation and the length of the chromosome. If too few bits are used, problems of insufficient accuracy may arise. On the contrary, if too many bits are used the chromosomes become exceedingly long which leads to a loss of efficiency in the evolutionary algorithm (Whitley et al. 1990). As a measure to circumvent loss of precision in representation, real numbers are used to represent connection weights. In addition, by using a vector of real values in representing all the connection weights of a neural network, direct manipulation of the connection weights can be achieved. Perhaps a better way to evolve a population of real-valued vectors is to use evolution strategies or evolutionary programming that is more suited to optimisation problems with continuous values. If the representation is vectors of real numbers, a crossover operation only creates new combinations of current connection weights. However, mutation actually creates new values of connection weights that differ from the initial set of connection weights. Furthermore, mutation also avoids the problem of producing offsprings that are exactly the same as their parents. Successful applications using evolutionary programming or evolution strategies evolving connection weights with real-valued representations can be found in Porto et al. (1995), Fogel et al. (1995), Yao et al. (1996), Greenwood (1997), Sarkar and

Yegnanarayana (1997), Chellapilla and Fogel (2001), Tesauro (2002) and Fogel et al. (2004).

2.4.3.2 Evolving Network Architecture

Evolving artificial neural network architectures can be viewed as a search through a space of all possible network structures (connectivity and the activation function). Algorithm 2.7 shows a typical algorithm for evolving network architectures. The process stops when a satisfactory artificial neural network is found.

1- Each hypothesis of network architecture in the current generation is encoded into chromosomes for genetic operations, by means of a direct encoding scheme or an indirect encoding scheme.

2- Evaluation of fitness. Decode each individual in the current generation into architecture, and build the corresponding ANNs with different sets of random initial connection weights. Train the ANNs with a predefined learning rule, such as the Backpropagation algorithm. Compute the fitness of each individual (encoded architecture) according to the training results, for example, mean-square-error, and other performance criteria such as the complexity of the architecture, e.g., less number of nodes and connections preferred.

3- Select parents from the population based on their fitness.

4- Apply search operators to the parents and generate offspring, which form the next generation.

Algorithm 2.7 A typical cycle for evolving network architectures in evolutionary artificial neural networks(Yao 1999a).

A direct encoding scheme for network architectures specifies all the details of the architecture in a chromosome, i.e. every connection and node of the architecture (Whitley et al. 1990; Fogel 1993; McDonnell et al. 1994). Following the encoding of the network architecture into binary strings, the evolution of the population of encoded architectures is obtained by employing crossover and mutation operators. As mentioned in the previous section, crossover operations may lead to inefficiency in the evolution of network

architectures which shows itself in several ways. First, artificial neural networks, as described in section 2.4.1, are a distributed representation form of knowledge, that is, each node and connection weight of the network acts as a storage point for the knowledge of solving a problem. One single node or connection does not explain any useful knowledge about the complete problem. Instead, using a cluster of hidden nodes with a set of connection weights, are used to discover and extract certain features from the inputs in a way that is comparable to the brain which can be divided into different regions with specified functions. During the evolutionary process, crossover operators are more likely to obliterate these useful feature detectors than the mutation process. Secondly, crossover operators suffer from the negative effect resulting from a permutation problem. This happens when two artificial neural networks order their hidden nodes differently but still have equivalent functionality (Hancock 1992; Igel and Stagge 2002). In general, crossover is not used as the principal operator in most evolutionary artificial neural network applications (McDonnell and Waagen 1994; Heimes et al. 1997; Fang and Xi 1997; Yao 1997; Yao and Liu 1997b). Hancock (1992) and Likothanassis (1997) argued that crossover might be imperative for some problems. Stanley and Miikkulainen (2002) showed increased efficiency on benchmark Reinforcement Learning tasks using their method of crossover on different network topologies. Further research is required to understand the efficiency of crossover operators in evolving artificial neural networks. With regard to direct encoding of network architectures, one of the issues is the length of the chromosome. As the size of the network grows, the length of the chromosome increases thus reducing the efficiency of the evolutionary

algorithm. With an indirect encoding scheme, there is a tendency to decrease the length of the genotype representation of architectures and only some characteristics of the architecture are encoded (Kitano 1990; Harp et al. 1990; Gruau 1994; Grönroos et al. 1999). For example, a parametric representation may only contain a set of parameters such as the number of hidden layers and the numbers of hidden nodes in each layer, assuming the networks are all feed forward multi layer perceptrons (Harp et al. 1990). Apparently, this greatly restricts the choice of potential network architectures. Development rule representation is another well-known indirect encoding scheme. It encodes development rules in chromosomes (Kitano 1990). These development rules specify certain primary building blocks in a network. The evolution of architectures is transferred to the evolution of development rules. The development rule representation can diminish the damaging effect of crossover although extra effort is needed during the encoding and decoding of chromosomes. However, development rule representation seems to be inefficient at evolving detailed connectivity patterns amongst individual nodes. Another downside of development rule representation is that it separates the evolution of architectures and the evolution of connection weights, which renders it inappropriate for the simultaneous evolution of architecture and connection weights. For more discussions on indirect encoding of network architectures, please refer to Moriarty and Mikkulainen (1997) and Yao (1999a).

2.4.3.3 Simultaneous Evolution of Architecture and Weights

The evolution of network architectures, as a distinct process, from the evolution of connection weights is described in algorithm 2.10. This separation could give rise to noise problems in the fitness evaluation of individual architecture hypothesis (Yao and Liu 1997a). Random initialisation of connection weights, when the individual architectures are evaluated, is the first source of the noise due to the fact that different random initial weights may generate different training outcomes. The training algorithms used for the evaluation creates the second source of noise. Even with the same set of initial weights, various training algorithms may generate various training results. To address these problems, simultaneous evolution of both the architecture and weights is recommended. There have been a number of studies on evolving architectures and connection weights simultaneously. An evolutionary system called NEAT (Stanley and Miikkulainen 2002 and Stanley 2006) was originally developed to solve difficult control and sequential decision tasks. NEAT is based on three principles that work together to efficiently evolve network topologies and weights. The first principle is homology: NEAT encodes each node and connection in a network with a gene. Whenever a structural mutation results in a new gene, that gene receives a historical marking. Historical markings are used to match up homologous genes during crossover, and to define a compatibility operator. Figure 2.7 shows A NEAT genotype to phenotype mapping example, while figure 2.8 shows the two types of structural mutation in NEAT.

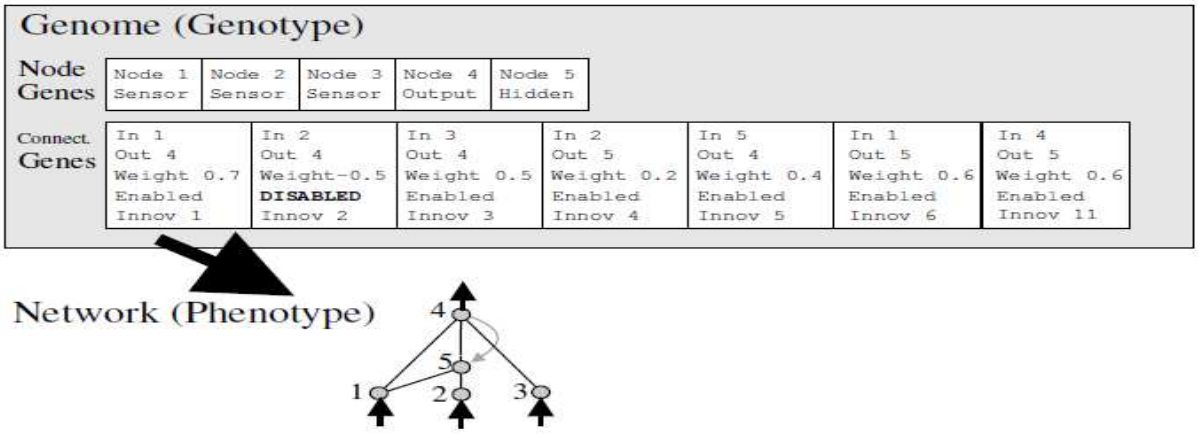

Figure 2.7 A NEAT genotype to phenotype mapping example (**Stanley and Miikkulainen 2002)**. A genotype is depicted that produces the shown phenotype. There are 3 input nodes, one hidden, one output node, and seven connection definitions, one of which is recurrent. The second gene is disabled, so the connection that it specifies (between nodes 2 and 4) is not expressed in the phenotype. In order to allow complexification, genome length is unbounded.

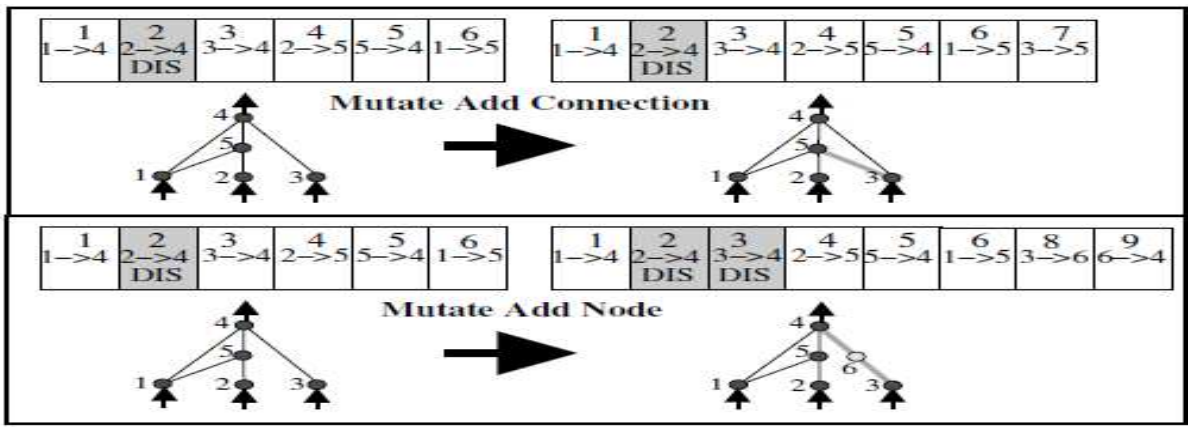

Figure 2.8 The two types of structural mutation in NEAT (Stanley and Miikkulainen 2002). Both types, adding a connection and adding a node, are illustrated with the genes above their phenotypes. The top number in each genome is the *innovation number* of that gene. The bottom two numbers denote the two nodes connected by that gene. The weight of the connection, also encoded in the gene, is not shown. The symbol DIS means that the gene is disabled, and therefore not expressed in the network. The figure shows how connection genes are appended to the genome when a new connection and a new node is added to the network. Assuming the depicted mutations occurred one after the other, the genes would be assigned increasing innovation numbers as the figure illustrates, thereby allowing NEAT to keep an implicit history of the origin of every gene in the population.

The second principle is protecting innovation. A compatibility operator is used to speciate the population, which protects innovative solutions and prevents incompatible genomes from crossing over. Finally, NEAT follows the philosophy that search should begin in as small a space as possible and expand gradually. Evolution in NEAT always begins with a population of minimal structures. Structural mutations add new connections and nodes to networks in the population, leading to incremental growth. Topological innovations have a chance to realise their potential because they are protected from the rest of the population by speciation. Because only useful structural additions tend to survive in the long term, the structures being optimised tend to be the minimum necessary to solve the problem. NEAT's approach allows fast search because the number of dimensions being searched is minimised. Figure 2.9 shows the matching up of genomes for different network topologies using innovation numbers.

Another important part of artificial neural network architecture is the activation function. In a design described by White and Ligomenides (1993), node activation functions are evolved using sigmoid and Gaussian functions in different ratios. Node activation functions are evolved by setting 80% of the nodes in the initial population using a sigmoid function and using Gaussian function to set the remaining 20%. The evolution seeks to establish the optimal mixture between these two activation functions.

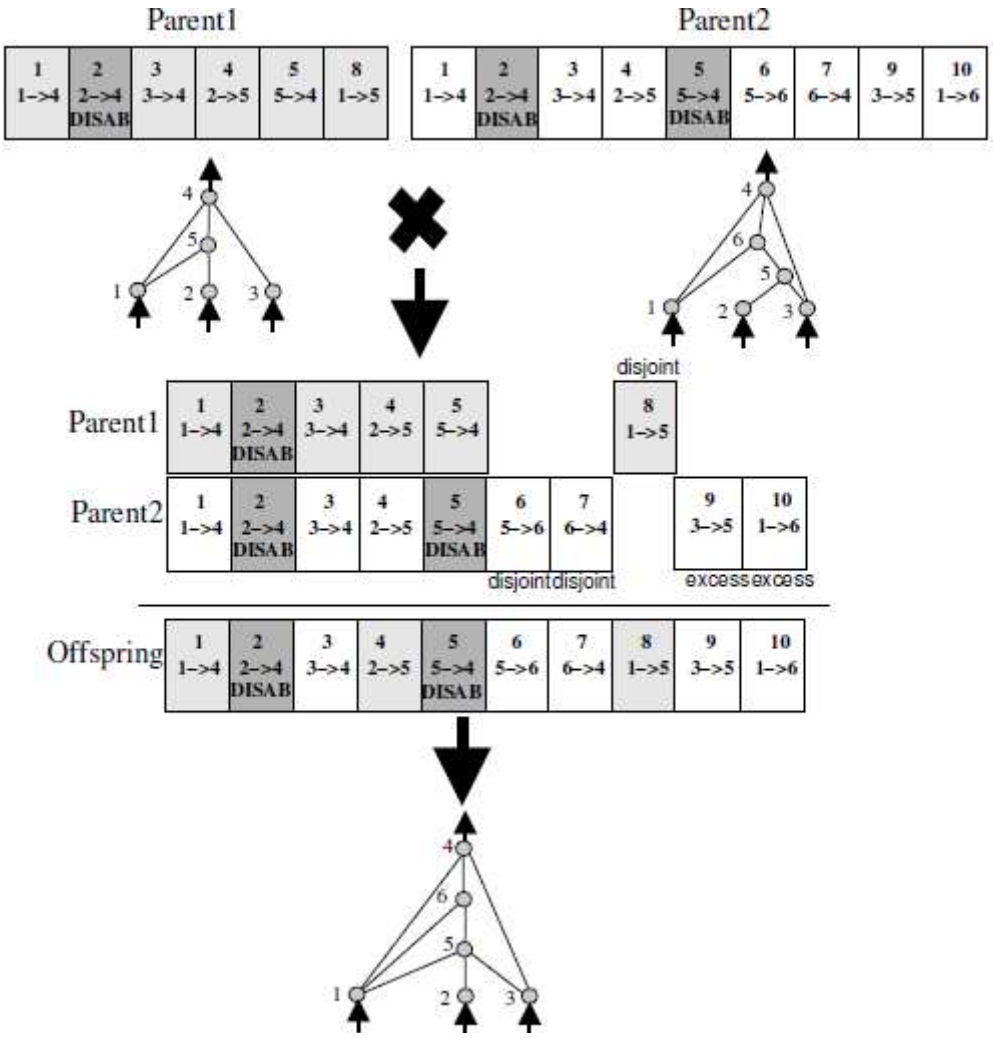

Figure 2. 9 Matching up genomes for different network topologies using innovation numbers (Stanley and Miikkulainen 2002). Although Parent 1 and Parent 2 look different, their innovation numbers (shown at the top of each gene) tell us that several of their genes match up even without topological analysis. A new structure that combines the overlapping parts of the two parents as well as their different parts can be created in crossover. In this case, equal fitnesses are assumed, so each disjoint and excess gene is inherited from either parent randomly. Otherwise the genes would be inherited from the more fit parent. The disabled genes may become enabled again in future generations: There is a preset chance that an inherited gene is enabled if it is disabled in either parent.

2.4.3.4 Evolving Learning Rules

In addition to learning rules such as backpropagation for multi layer perceptrons, other types of learning rules for different types of artificial neural networks also exist, such as the Hebbian learning rule (Fausett 1994). In fact, we can assume any learning rules to be in a more general form as follows (Mitchell 1999):

$$W_{ji}(t+1) = W_{ji}(t) + \Delta W_{ji}$$

Where

$$\Delta W_{ji} = f(a_i, o_j, t_j, w_{ji})$$

a_i is the input to unit *i*. o_j is the output from unit *j*. t_j is the targeted output from unit *j*. w_{ji} is the current weight on the connection from *i* to *j*. We can assume the learning rule f to be a linear combination of these variables. Examples of evolving learning rules can be found in Chalmers (1990) and Baxter (1992).

2.5 COMPUTER GAME PLAYING

Designing automated computer game playing programs has been of interest since the 1950s (Turing 1950; Samuel 1959), and is still of interest today, with successes such as Deep Blue in 1997 (Newborn 1997; Campbell et. al. 2002), which defeated Garry Kasparov, considered the best ever chess player. Game playing involves many important aspects of interest to artificial intelligence such as knowledge representation, search and machine learning. Traditional computer games programs use a knowledge based approach, where human knowledge about the game is encoded by hand into the

computer by means of an evaluation function and a database of opening and end game sequences.

In 1954, Arthur Samuel developed a checkers player in an attempt to demonstrate that a computer program could improve by playing against itself. Samuel's program adjusted weights for 39 features (Samuel 1959, 1967). Samuel used a form of what is now called "reinforcement learning" (to find more about reinforcement learning, please refer to Kaelbling et al. 1996; Mitchell 1997; Sutton and Barto 1998; Vrakas and Vlahavas 2008) to adjust these features, instead of tuning them by hand. Samuel discovered that the most important feature was the piece difference and the remaining 38 features (including capacity for advancement, control of the centre of the board, threat of fork, etc.) varied in their importance. Due to memory limitations Samuel used only 16 of the 38 features in his evaluation function, swapping between them to include the remaining 22, which he called term (Samuel 1959, 1967; Fogel 2002). Two evaluation functions (alpha and beta) were used to determine the weights for the features. At the start, both alpha and beta have the same weight for every feature. Alpha weights were modified during the execution of the algorithm. Beta values remained static. The process gave an appropriate weight to each feature when summed together. Each leaf node in the game tree was evaluated using this evaluation function. This process represents one of the first attempts to use heuristic search methods in searching for the best next move in a game tree. Samuel (1959) used minimax with three ply search and a procedure called *rote learning*. This procedure was responsible for storing the evaluation of different board positions in a look-up table for fast retrieval (Look-Ahead and

memorization). Samuel (1967) improved the minimax search with alpha-beta pruning that incorporated a supervised learning technique to allow the program to learn how to select the best parameters to be calculated in the evaluation function. In July 1962 Samuel's program played against Robert Nealey, described (incorrectly) as a former Connecticut checkers champion, and one of the nation's foremost players. Samuel's program defeated Nealey, the first time a computer program had defeated a state champion (although he earned this title four years later). At that time it was considered a great success and a significant achievement in machine learning. In fact this was the only win that Samuel's program managed against Nealey, or any other players, and there is some controversy about how strong a player Nealey really was. Samuel claimed that his program focused on the problem of having a machine learning program, rather than be told how to play, but in fact he used 39 features (although he wanted to get away from that requirement), which some would argue is utilising human knowledge. However, the historical importance of this work cannot be underestimated as it set the challenge which Fogel was later to accept, and to answer.

In 1989, Jonathan Schaeffer and his colleagues at the University of Alberta, designed a checkers program called Chinook (Schaeffer et al. 1996; Schaeffer 2009), which later became the world champion at checkers. Schaeffer's initial motivation was to *solve* the game. However, this was a challenging goal as there are approximately $5*10^{20}$ different positions to evaluate (Schaeffer 2009). A further motivation was to produce the world's best checkers player. This was done by using an evaluation function, which comprises several features, all of them based on human expertise, including

grand masters. The main feature in Chinook's evaluation function is the piece count, where each piece on the board takes 100 points. The next most important feature is the king, which takes a value that is greater than a regular checker by 30 percent, except when the king is trapped (a trapped king cannot move because it will be taken by the opponent), when it takes the same value as a regular checker. Another feature that is important to Chinook's evaluation function is the *runaway* checker (a clear path for a checker to become a king, without any obstacles), which takes a value of 50 points in addition to its previous value, and subtracts three points for each move that is required to advance the checker to be a king. There are other additional features that are included in the evaluation function, including the "*turn*", "*mobile kings*" and the "*dog hole*" (a checker that is trapped by its opponent and cannot be moved). Each one of those features was assigned a different weight indicating its importance. The summation of each term provided an overall assessment of the board for that particular game state, which enabled different game states to be compared. Initially, Schaeffer gave initial values to the weights and then hand tuned them when he found an error (e.g. an obviously incorrect move being made) or when a Chinook move led to position that led to a losing position. Chinook also utilised opening and end game databases to further enhance its ability. Initially Chinook's opening game database comprised of 4,000 sequences. Later it contained more than 40,000. The end game database contained all the possibilities that lead to a win, a draw or a loss, for a given number of pieces left on the board. The final version of Chinook's end game database contained all six piece end sequences, allowing it, together with the ability to

determine the right move, to play perfectly from these positions. In 1989 Chinook, with a four-piece end game database (Schaeffer et al. 1992), won the computer Olympiad. Later, with its final six-piece end game database, together with its evaluation function modified by a *fudge* factor (Schaeffer et al. 1993; Schaeffer 2009), it finished in second place to Marion Tinsley (recognized as the best checkers player who ever played the game) in the U.S. National Checkers Championship held in 1990. After a further sequence of matches in 1994 between Chinook and Tinsley, Chinook became the world man machine checkers champion (after Tinsley's resignation due to health problems, he died the following year) (Schaeffer 2009). In 1996 Chinook retired with a rating of 2,814. The building of the open/end game databases ultimately led Schaeffer to achieve his initial motivation (solving the game of checkers) (Schaeffer et al. 2007). Perfect play by both sides leads to a draw.

Neurogammmon (Tesauro 1989) is computer program that learns how to play backgammon. Neurogammon uses a multilayer feed forward neural network that was trained on a large data set obtained from human experts. The training was carried out using a backpropagation neural network. Neurogammon used one network to make a *doubling cube* and another six networks that made ordinary moves. Each network is fully connected with one hidden layer. The input to the network was an initial board position and this board position also fed directly to the final position. The output was the experts' decision that judged the best move to make, given initial board positions. "*Comparison paradigm*", (Tesauro 1989), was used to teach the network how to favour moves that were made by the expert by giving it a score higher than that assigned to other moves. Neurogammon won the First

Computer Olympiad (held in London), with a record of five wins and no losses (Tesauro 1989). However, Neurogammon lost to a human expert, Ossi Weiner from Germany, with the final score being 7-2. Weiner commented that Neurogammon played like a human and only made a few mistakes, which was considered as a significant accomplishment for the program.

During the mid 1990s, IBM produced Deep Blue (Campbell et al. 2002) in an attempt to create a chess program that was capable of beating the world champion at that time. The history of chess computer programs, and early works of Deep Blue, is described in (Hsu et al. 1990; Goetsch and Campbell 1990; Newborn 1997; Heinz 2000; Hsu 2002). Deep Blue had 30 processors (Hsu 1999) that were able to carry out a parallel search, and could evaluate up to 200 million chess positions per second (Clark 1997). Deep Blue's evaluation function comprised about 8,000 different features. Each feature had a weight, which was initialised by the evaluation function generator (some features had static values). The evaluation function can be calculated by summing up those weights. The opening database in Deep Blue consisted of 4,000 positions that had been manually entered according to human grandmasters. A new technique, called "*Extended Book*" (Campbell 1999) was also used in Deep Blue, which was capable of extracting useful knowledge from over 700,000 grandmaster chess games. This information directed Deep Blue in its opening moves. The extended book evaluation function includes a number of factors. Among these were "The number of times a move has been played", "The relative number of times a move has been played", "Strength of the players that play the moves", "Recentness of

the move", "Results of the move", "Commentary on the move" and "Game moves versus commentary moves". These factors were combined in a nonlinear function to produce a scalar output value (as high as half value of a pawn). The end game database of Deep Blue consists of all positions with five or fewer chess pieces on the board, which is stored in a database as one bit per position (either lose or not). In order to keep control of the time, Deep Blue used two types of time settings, which had to be set before each search. The first one is the normal time that is set to be the time remaining to the next time control divided by the moves remaining, while the second time setting is the panic time, which is one third of the remaining time.

After losing against Gary Kasparov (World Chess Champion) in 1996, Deep Blue defeated Kasparov in a six-game match in 1997 to become the first computer program to defeat a world chess champion (note that Chinook had performed a similar feat for checkers three years earlier). King (1997) provides more insights to the 1997 match.

In 2006, a group of researchers presented MoGo (Gelly et al. 2006; Gelly and Wang 2006), a computer program that played the game of Go. The design of MoGo focused on two main elements. The first was to make a modification to the UCT (Upper bound Confidence for Tree) algorithm (Kocsis and Szepesvari 2006), while the second focused on using techniques such as parallisation, pruning and dynamic tree structure (Coulom 2006). The design of MoGo consisted of two phases; (1) the design of the tree search and, (2) a random simulation. The tree is created dynamically by adding one node after each simulation phase (used to evaluate the whole tree created so far). In

August 2006 MoGo was ranked top of 142 programs to play Go according to the classification of 9x9 Computer Go Server. MoGo also won all the tournaments which were held by the Kiseido Go Server during October and November 2006. The tournaments played matches on 9x9 and 13x13 Go boards. MoGo with Monte Carlo tree search reached the level of 3 Dan in Taiwan's Computer Go Tournament, 2008 (Lee et. al. 2009).

One of the criticisms of traditional knowledge-based approaches for developing game-playing machine intelligence is the large amount of pre-injected human expertise that is required for the computer program, together with the lack of learning capabilities of these programs (Fogel 2000; Fogel 2002). The evaluation functions and opening and end game databases described above are provided by game experts. In this sense, a computer game's *intelligence* is not gained by actually playing a game, but rather comes from the pre-designed evaluation function and a look up database of moves. Moreover, this intelligence is not adaptive. It could be argued that humans read books and watch other people playing a game before they actually start playing themselves. Humans also improve their skill through trial-and-error. New features and strategies for playing a game can be discovered by new players rather than grand masters, while old features could be viewed as worthless and old strategies are discarded. Humans also adapt their strategies when they meet different types of players, under different conditions, in order to accommodate their special characteristics. We do not see such adaptations and characteristics in the knowledge-based computer game programs. Fogel (2002) commented on this phenomenon in computer game-playing:

> *"… To date, artificial intelligence has focused mainly on creating machines that emulate us. We capture what we already know and inscribe that knowledge in a computer program. We program computers to do things – and they do those things, such as play chess, but they only do what they are programmed to do. They are inherently "brittle". … We'll need computer programs that can teach themselves how to solve problems, perhaps without our help*. …"

The following computer games are based on self learning techniques rather than the pre-injection of human expertise.

In 1998 Norman Richards and his colleagues from the university of Texas produced a self learning program (Richards et al. 1998) that was capable of playing the game of Go on small boards (9x9), without any injection of prior knowledge. This program used the SANE (Symbiotic Adaptive Neuro-Evolution) method (Moriarty and Miikkulainen 1998; Lubberts and Miikkulainen 2001) to evolve neural networks to be able to play Go on simple boards. The design of the neural network consisted of a three (two input and one output) layer feed-forward network with evolvable connection weights. The input units were used to indicate whether the black or white stones were present, while the output unit indicated whether a move is good or not (a positive value reflects a good move, while a negative or zero value indicates a bad move). The evaluation function of SANE used Chinese scoring by counting all the stones of the same color, together with all locations completely surrounded by stones of that color and the difference in the scores between SANE and its opponent is summed over *N* games and used

as a fitness level for the networks. SANE was tested by playing against Wally (written by Bill Newman), on a 5x5 board, SANE needed only 20 generations to defeat Wally, while it needed 50 generations on a 7x7 board. On a 9x9 board, SANE needed 260 generations.

Blondie24 (Fogel 2002) represents an attempt to design a computer checkers program, injecting as little expert knowledge as possible. Evolutionary neural networks were used as a self-learning computer program. The neural network used for a particular player provided the evaluation function for a given board position. Evolution strategies made these networks, which acted randomly initially (as their weights were initialised randomly), gradually improve over time. The final network was able to beat the majority (>99%) of human players registered on www.zone.com at that time. Blondie24 represents a significant achievement, particularly in machine learning and artificial intelligence. Although Blondie24 does not play at the level of Chinook (Schaeffer 2009), this was not the objective of the research; but rather to answer the challenges set by Samuel (1959, 1967) and also by Newell and Simon (two early AI pioneers). The next section (section 2.6) provides more details about the implementation of Blondie24 and discusses the results along with the perceived shortcomings.

TD-Gammon (Robertie, 1992; Tesauro 2002) represents a first attempt to produce a self learning computer program that is able to play a game of backgammon to the level that is competitive with human experts. TD-Gammon is a neural network based computer program that is able to teach itself how to play the game of backgammon by playing against itself starting

from completely random initial play. TD-Gammon used multilayer perceptron neural networks, which takes a sequence of board positions from the start, until the end (one side succeeds in removing all their pieces) and produces an output that represented the network's estimation about how good is that board position. No features were encoded in the neural network during training and the network was used to select the best move for both sides (learning from the results of playing against itself). TD-Gammon contained 160 hidden nodes and performed a three-ply search. It was trained for over six million self play games (Tesauro 1992, 1995). TD-Gammon has been tested against many human players during its different versions, with different modifications, and was shown to be very successful. TD-Gammon was also shown to be able to play better against human experts than Neurogammon (Tesauro 2002).

Blondie25 (Fogel et. al. 2004) (a development of Blondie24 but now for chess), was an attempt to produce a self learning evolutionary chess program that can learn how to play the game of chess by playing against itself, injecting as little expert knowledge as possible. Blondie25's implementation worked as follows: The chessboard was represented as a vector of length 64, where each component in the vector represents a board position. Components in the vector could take values from {-K, -Q, -R, -B, -N, -P, 0, +P, +N, +B, +R, +Q, +K} where 0 represented an empty square and the variables P, N, B, R, Q, and K represented material values for pawns, knights, bishops, rooks, and the queen and king, respectively. The sign of the value indicated whether or not the piece in question belonged to

the player (positive) or the opponent (negative). Three fully connected artificial feedforward neural networks were used, each one with 16 inputs, 10 hidden nodes, and a single output. The three neural networks focused on the first two rows, the back two rows and the centre of the chess board as shown in figure 2.10 (Fogel et. al. 2005). To start the evolutionary process, 20 computer players were initialised with the values 1,3,3,5,9 and 10,000 for P, N, B, R, Q and K respectively (Fogel et. al. 2005). Each player played 10 games (five as white and five as black) against 10 randomly selected players from the same population and according to their scores (+1 for win, 0 for draw and -1 for lose) the 10 players which scored more points were selected and the others were killed off. The selected players were mutated to produce 10 offspring. The best player from the last generation was selected to be Blondie25. Games were played using an alpha beta search with a four ply depth. Blondie25 was tested against many popular chess programs (Fogel et. al. 2006) and showed success in defeating Fritz 8, ranked number 5 in the world at that time. Also Blondie25 defeated a human master, ranked 2301 at that time. Blondie25 itself is ranked at about 2640.

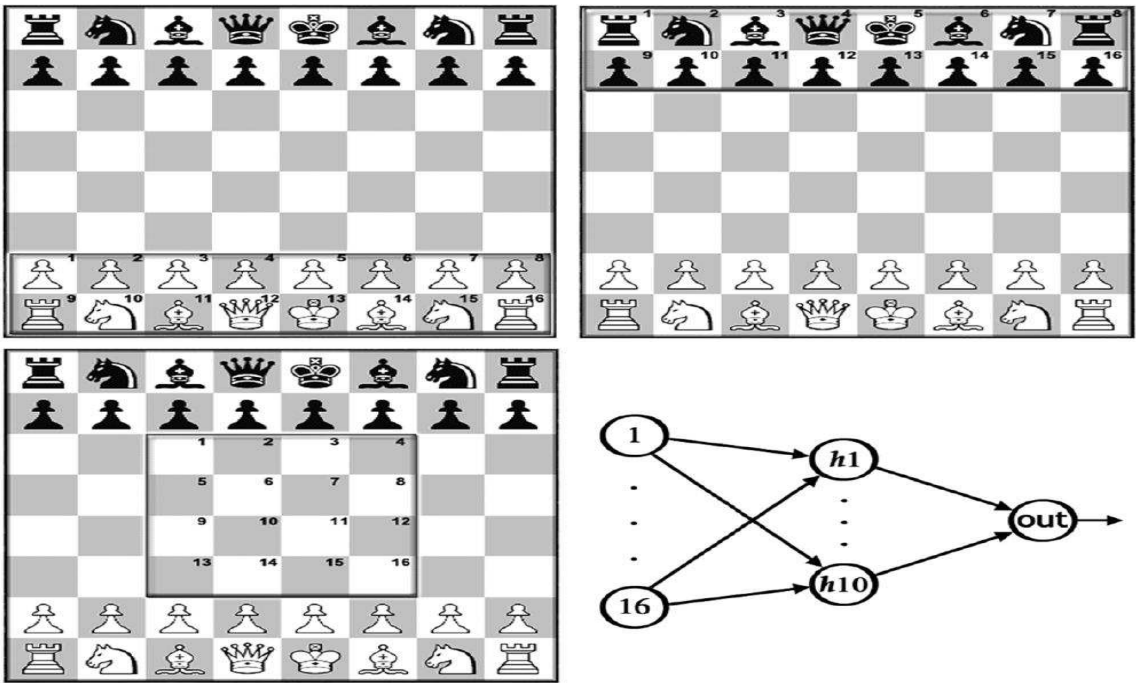

Figure 2.10 The three chess board positions (Fogel et al. 2004).

We can find many other studies and applications in game playing in recognition of intelligence as an evolutionary process, such as Turing (1950), Fogel et al. (1966), Axelrod (1987), Fogel (1992), Fogel (1993), Kendall and Hingston (2004) for the Iterated Prisoner's Dilemma, Moriarty and Miikkulainen (1995) for the game of Othello, Pollack and Blair (1998) for the game of Backgammon, Richards et al. (1998) and Kendall et al. (2004) for the game of Go, and Kendall and Whitwell (2001), Baxter et al. (2001), Stanley et al. (2005) for Nero and Nasreddine et al. (2006) for Chess together with Lucas and Kendall (2006) for various computational intelligence methods that can be used for various games.

2.6 BLONDIE24

In 2000, an evolutionary algorithm was presented by David Fogel which was capable of playing the game of checkers, injecting as little expert knowledge as possible. By solely using the number, type and positions of pieces on the checkers board, the evolutionary program utilises feed forward artificial neural networks to evaluate alternative positions in the game. Fogel called his evolutionary program Blondie24 (Fogel 2002). Blondie24 is a checkers program that is capable of learning how to play checkers to a level close to that of human experts. In comparison, the major difference between Blondie24 and other traditional game-playing programs is in the employment of the evaluation function (Chellapilla and Fogel 2001; Fogel and Chellapilla 2002). In traditional game-playing programs, evaluation functions usually comprise important features derived from expert human techniques for generating good moves. Hand tuning is used to alter the weighting of these features. In Blondie24, the evaluation function is an artificial neural network that only knows the number of pieces on the board, the type of each piece and their positions; no other inputs such as human experience about the techniques of the game, are pre-programmed into the neural network.

2.6.1 Blondie24 Implementation

As mentioned above, the core feature in the design of Blondie24 is to make the program learn, through self play, how to play checkers. This is in direct contradiction of an alternative which is to preload it with all the information about how to make good moves and avoid bad ones (Chellapilla and Fogel 2000; Fogel 2000). The design of Blondie24 program consists of

two main modules: The artificial neural network and the checkers engine module (Chellapilla and Fogel 1999, 2000). Each designed module consists of sub-modules that are designed to achieve certain tasks.

2.6.1.1 The Artificial Neural Network Module

This module concerns the design of the Evolutionary Artificial Neural Network (EANN) that will be used as an evaluation function for the current checkers board position. The EANN takes a vector of length 32 as input, with each element representing an available position on the checkers board (checkers is only played on half the available squares on an 8X8 board) and produces a scalar output ranged [-1, +1]. A value of +1 represents the value of a winning board and -1 represents the value of a losing board. Values between -1 and +1 demonstrate how good the board is at this particular point (the higher the better). Components in the input vector take elements from {-*K*, -1, 0, +1, +*K*}, where 0 corresponds to an empty square, 1 is the value of a regular checker, and K is the number assigned for a king. Initially *K* was set to 1.5 (Fogel 2002). The sign of the value indicated whether or not the piece belonged to the player (positive) or the opponent (negative). The evaluation function was structured and implemented as a feed forward neural network with an input layer, three hidden layers, and an output node. The second and third hidden layers (comprising 40 and 10 units respectively) and the output layer had a fully connected structure. The first hidden layer connections were specifically designed to capture spatial information from the board. The 8x8 checkers board was converted to a 1 x 32 vector as input to the first hidden layer, which consisted of 91 pre-processing nodes which

captured the spatial characteristics of the board. These 91 nodes covered a variety of *n* x *n* squares overlapping subsections of the board. The reason to choose these *n* x *n* subsections was to provide spatial adjacency or proximity information such as whether two squares were neighbours, or were close to each other, or were far apart. To the first 36 hidden nodes in the first hidden layer, all the 36 possible 3 x 3 square subsections of the board were supplied as input. The following 25 4 x 4 square subsections were assigned to the next 25 hidden nodes in that layer. The 16 5 x 5 square subsections were assigned to the next 16 hidden nodes. The 9 6 x 6 square subsections were assigned to the next 9 hidden nodes. The 4 7 x 7 square subsections were assigned to the next 4 hidden nodes. Finally the entire board (8 x 8 square subsections) was assigned to the last hidden node in that layer. All possible overlapping squares of sizes 3 to 8 were given as inputs to the 91 nodes of the first hidden layer. This made the neural network able to produce features from these entire board subsets that could then be processed in subsequent hidden layers (of 40 and 10 hidden units). Figure 2.11 illustrates the general structure of the neural network. Any inclusion of an expert's experience was avoided in the design of Blondie24, which was attempting to achieve Samuel's challenge (Fogel 2002) concerning the level of play that could be obtained simply by using evolution to extract linear and nonlinear features about the game of checkers and to optimize the interpretation of those features within the neural network without using any human expertise, making the computer able to learn these features on its own. The only exception was achieved by providing the neural network with a piece differential that connected directly to the output node (Chellapilla and Fogel

1999; Fogel 2002). This originated from the fact that even novice players would recognize which side had more pieces.

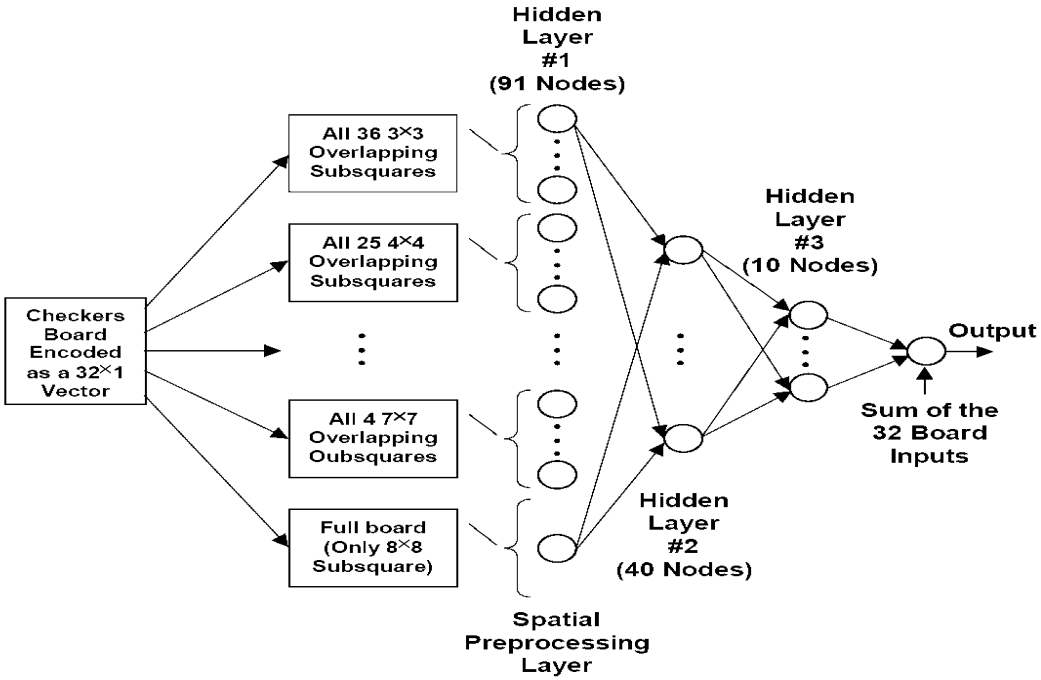

Figure 2.11 EANN architecture(Fogel 2002).

The nonlinearity function at each hidden node and output node was the hyperbolic tangent (Fogel 2000 and Chellapilla and Fogel 2001) (bounded by ±1), which can be implemented as follows:

Activation = (Exp(value) - Exp(-value)) / (Exp(value) + Exp(-value))
Where, value is the summation of the dot product between the inputs and corresponding weights in the node.

2.6.1.2 Checkers Engine

The design of this module consists of three sub-modules (Fogel 2002). The first, and most important, is the actual checkers playing sub-module which is used to record all the information about the board and the checkers pieces. It

also indicates legal moves (ordinary moves, jump moves). The checkers engine requires a search algorithm that generates a tree of all valid moves at any given board positions and then applies the neural network module to evaluate the leaf nodes and then, by using alpha-beta cutoff, propagate these values back up the search tree in order to choose the best available move. Blondie24 used depth first search to expand the search space to a certain depth (usually 4 or 6). For each move, the search can be made by examining the checkers board from the top left corner taking into consideration every available piece on board. If a valid move is found then the search is extended by examining the opponent's valid moves by using the same process. This process continues until the maximum play level is reached. The search space is extended every time a jump move is discovered, until no further jumps are available. The search stops below any discovered jump move and no further expansion of other valid moves is performed. This was done to adhere to the rules of checkers. The checkers Playing Sub Module takes two players red and white (two EANNs, *i* for red and *j* for white) and plays checkers in the following way: the Search sub module is called to produce a search space tree to the current depth *d.* The leaf nodes are then evaluated using the EANN currently being used. These values are propagated back to the root of the tree utilising the Alpha-Beta Pruning Sub Module, in order to decide the best move to play.

2.6.2 The Evolutionary Process

The evolutionary algorithm is started by initialising a population of 30 neural networks P_i, i= 1, ..., 30. These networks are called strategies

(Chellapilla and Fogel 1999, 2001). Each strategy is created randomly by assigning weights and biases in the range [-0.2, 0.2]. An associated self-adaptive parameter vector s_i, i = 1, ..., 30 is set for each neural network. These vectors are initially set to 0.05 to be consistent with the range of initialisation (Chellapilla and Fogel 1999, 2001). The associated self-adaptive parameter is used to control the step size of the search for mutated parameters of the neural network. All the neural networks are put into a competition with one another. Five games of checkers are played by each neural network as a red player with points being received for their resulting play. The five opponents (playing as white) are randomly selected to play against each red player. In each game, the red player and the white opponent scored -2, 0, or +1 points depending on whether it lost, draw, or won the game, respectively. A draw was declared after 100 moves for each side. The reason to choose these values was to try to make the player avoid losing as much as possible. In total, there were 150 games per generation. After all the games were complete, Blondie24 retains the 15 neural networks that received the highest points total and uses them as parents for the next generation. The other remaining 15 neural networks, with the lowest scores, were killed off. To start the next generation, each parent of the 15 selected neural networks generated an offspring neural network with all weights and biases being mutated. Specifically, for each parent P_i, i = 1, ..., 15 an offspring P'_i, i = 1, ..., 15, was created by:

$$s_i(j) = s_i(j)\exp(\, tN_j\,(0,1)\,),\ j = 1, \ldots, N_w \qquad (2.3)$$

$$w_i(j) = w_i(j) + s_i(j)N_j(0,1),\ j = 1, \ldots, N_w \qquad (2.4)$$

where N_w is the number of weights and biases in the neural network (here this is 5046), t= 1/sqrt(2sqrt(N_w)) = 0.0839, and $N_j(0,1)$ is a standard Gaussian random variable resembled for every *j*. The offspring king value *K'* was obtained by: *K'* = *K* + *C*, where *C* was chosen uniformly at random from {–0.1, 0, 0.1}. With the range of *K* being [1.0, 3.0] (Chellapilla and Fogel 1999; Chellapilla and Fogel 2000; Fogel 2000).

2.6.3 Results

The evolutionary process was iterated for 840 generations, which took about six months. The best evolved neural network was used as the final player, and called Blondie24. It played against human opponents on www.zone.com. The standard checkers system rating, which is the same as used for chess, was used to rate the players at this site. Initially, each player has a ranking of R_0=1600. The score for each player can be updated depending on the result of each game and the rating of the opponent as follows:

$$R_{new}=R_{old}+C(\text{outcome-W}) \quad (2.5)$$

where $W=1/(1+10^{((Ropp-Rold)/400)})$, R_{opp} is the opponent's rating, and C= 32 for ratings less than 2100, C = 24 for ratings between 2100 and 2399, and C = 16 for ratings at or above 2400 Outcome = {1 if Win, 0.5 if Draw, 0 if Loss}(Chellapilla and Fogel 1999; Chellapilla and Fogel 2001).

Blondie24 played 165 (84 as red and 81 as white) games against human players on www.zone.com. These 165 games were played over a two month period. No opponent was told that they were playing against a computer

program. When playing against players ranked below 2000 (in www.zone.com) Blondie24 won, lost, drew; 84, 20, 11 games respectively. However, when playing against expert opponents rated between 2000 and 2200, Blondie24 won 10, 12 drew and lost 22 games. Figure 2.12 and figure 2.13 show that after 165 games, Blondie24's average rating was 2045.85 with a standard deviation of 33.94, which put Blondie24 in the top 500 of the registered players on zone.com (better than 99.61% of the players registered on that website) at that time. Blondie24 was also tested by playing against Chinook (current world champion checkers program rated 2814) at the novice setting and won.

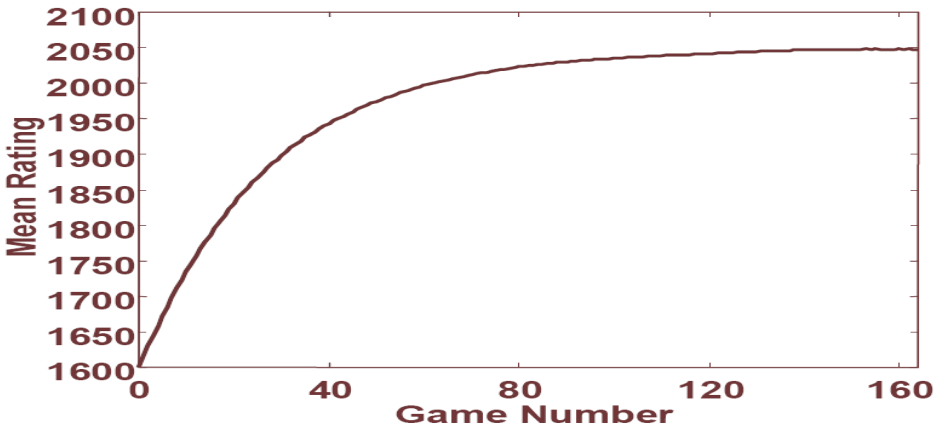

Figure 2.12 Blondie24 rating after 165 games on zone.com (Chellapilla and Fogel 2001).

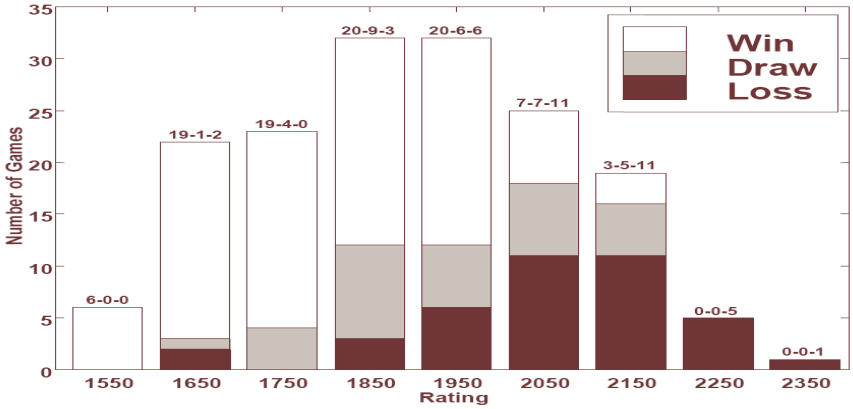

Figure 2.13 Blondie24 Performance after 165 games on zone.com (Chellapilla and Fogel 2001).

2.6.4 Discussion

Blondie24 represents a milestone in evolutionary learning but the evolution did not allow for the end product to learn any further (i.e. learning was only exercised in the evolution phase and no learning took place in the playing phase). This makes Blondie24 incapable of adapting itself when interacting with human players. Harley comments on this fact in his book review (Harley 2002):

> "… *An interesting point is that the end product which looks intelligent is Blondie, yet she is not in fact the intelligence. Like the individual wasp, Blondie is fixed in her responses. If she played a million games, she would not be iota smarter. In this sense, she is like Deep Blue. … Perhaps a better example of intelligence would be … a human, who can adapt her behavior to any number of new challenges*…"

To be more accurate, the creation of Blondie24 is to be considered as a learning process (achieving Samuel's challenge (Samuel 1967) but Blondie24 itself is unable to learn from its environment (Kendall and Su 2007).

2.7 INDIVIDUAL AND SOCIAL LEARNING

Inspired by Su (2005), this section will present individual and social learning in the context of game playing. In these discussions, the structure of individual and social learning in an imperfect evolutionary system will particularly be focused on. According to Su (2005), an imperfect evolutionary

system is "*a system where intelligent entities optimise their own utilities with the resources available whilst adapting themselves to the new challenges from an evolutionary imperfect environment*". To develop an imperfect evolutionary system, an integrated concept of individual and social learning has been employed. Four blocks participate in the formation of an imperfect evolutionary framework; namely, the imperfect environment, the imperfect individuals, individual learning mechanism and social learning mechanism. Adapted from Su (2005), brief descriptions of each of these blocks are stated below:

• The Imperfect Environment

- This is pivotal for the implementation of the imperfect evolutionary systems. The environment is made available by supplying information and knowledge for survival as well as acting as medium for evolution.

• The Imperfect Individuals

- Through individual learning, the imperfect individual exploits the available resources. Utilising social learning process, the individual attracts new information from the imperfect environment and gains better information and knowledge.

• Individual Learning Mechanism

- An evolutionary process where the individual optimises its own utilities.

• Social Learning Mechanism

- An evolutionary process where there is a process of learning from each other amongst all participants of an imperfect environment. Alongside, the information and knowledge distribute broadly within the imperfect evolutionary system.

In a model described in (Kendall and Su 2003), a stock market was used as a problem domain to evaluate the imperfect evolutionary systems. Here, an integrated individual and social learning mechanism was utilised by stock traders, to learn how to trade the stocks. Figure 2.14 shows a model of multi-agent based simulated stock market (Kendall and Su 2003).

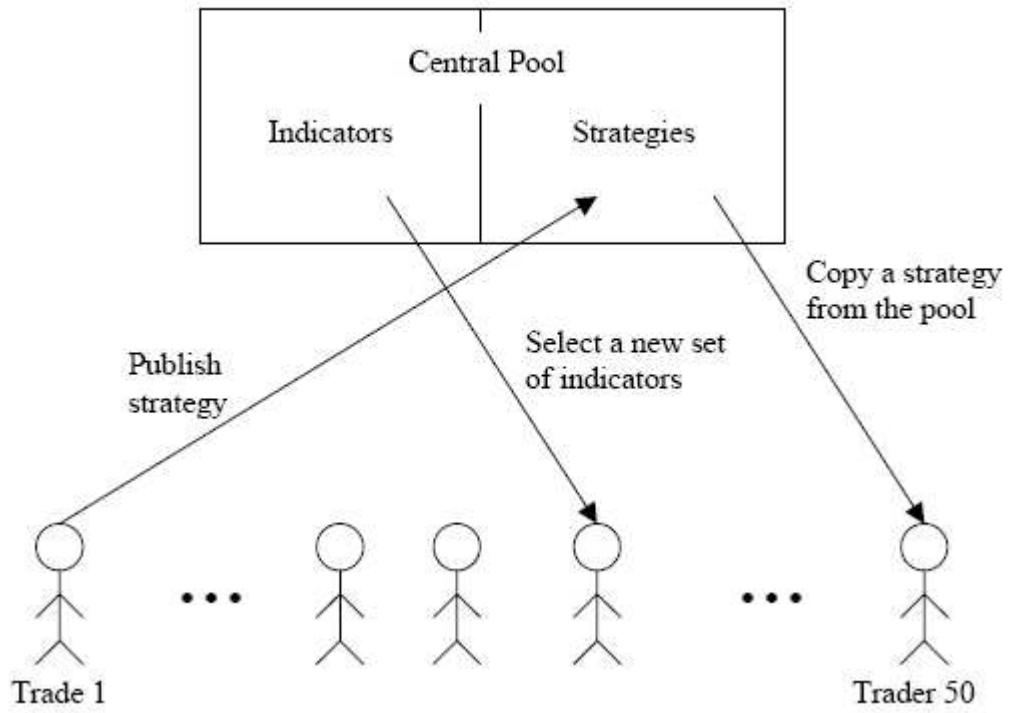

Figure 2.14 Model of a multi-agent based Simulated Stock Market (Kendall and Su 2003).

The general model is as follows (based on Figure 2.14 adapted from Kendall and Su 2003):

1. There are 50 traders before trading starts.

2. There are 20 indicators, each is assigned with value 1, and there are zero trading strategies in the central pool. Each trader selects a random set of indicators as inputs to their trading models.

3. Each trader generates 10 different artificial neural network models for forecasting based on selected indicator(s). These ten models may have different network architectures, but they use the same set of indicators selected by the trader. The aim is for the trader to evolve models from these ten by the means of individual learning.

4. The experiment is divided into 30 intervals where the total time span is 3750 trading days. Each interval has 125 days (6-month trading).

5. Each 125-day trading is divided into 25 intervals. At the end of individual learning (after 5 days for each interval), evolve the neural networks using EANNs with evolutionary programming.

6. Social learning occurs at the end of 125-day trading, where each trader has an opportunity whether to copy a better strategy from the central pool or publish its own strategy into the central pool.

7. The system enters the next interval after social learning finished and repeat steps 5 and 6.

Individual learning occurs during every 125-day trading period. Each trader builds their 10 prediction models based on the selected indicators. These ten models evolve using evolutionary programming. The general algorithm for individual learning is as follows (adapted from Su 2005):

- Select a neural network to be eliminated.
- Select a neural network for mutation using roulette wheel selection.
- Decide number of connections to be mutated, m, where m is

$$m_{offsbring} = m_{parent} + \sigma_m \quad (2.6)$$

Where, σ is a random Gaussian number with mean of zero and standard deviation of 0.1.

- set i = 0.
- While (i < m)

– Select the connection randomly.

– weight = weight + Δw, where Δw is a random Gaussian number with mean zero and standard deviation of 1, and Δw is also generated a new for each mutation.

– i = i + 1.

- With 1/3 probability, add a hidden node and randomly generate new connections.
- With 1/3 probability, delete a hidden node and delete all connections to it.
- Replace the network to be eliminated with the mutated neural network.

All traders enter social learning at the end of 125-trading day. At this stage, all traders compare their performance based on their self-assessment, where trader i rate of profit (ROP) in percentage is calculated using the following equation:

$$ROP_i = \frac{W_t - W_{t-5}}{W_{t-5}} \times 100 \quad (2.7)$$

Where,

- Wt is the value for trader's current assets (cash + shares).
- *Wt−5* is the value of trader's assets one week before.

Traders are ranked from 0 to 49 based on their ROP for the previous 125-days trading (six months):

$$S_{peer}^{i} = 1 - \frac{R_i}{49} \quad (2.8)$$

Where,

- R_i is the rank of trader i in the range of [0, 49] (0 indicates highest rank with greatest ROP).

The score from equation 2.8 shows trader i's performance compared to other traders. The following equation is used to calculate the performance of each trader for the past six months.

$$S_{self}^{i} = \frac{ROP - ROP'}{100} \quad (2.9)$$

Where,

- ROP is rate of profit for the current six months trading.
- *ROP'* is rate of profit for the previous six months.

Based on equations 2.8 and 2.9, the overall assessment for trader i is as in the following equation

$$assessment_i = S_{peer}^{i} + \frac{1}{1 + e^{1 - S_{self}^{i}}} \quad (2.10)$$

Algorithm 2.8 Individual Learning (Su 2005).

The activities in social learning are selected based on the normalisation on the overall assessment where they are normalised between 0 and 1. Social learning algorithm based on normalised value is as follows:

1. If normalised value is 1 and trader is not using a strategy drawn from the pool,
   - Publish the strategy into the pool and use the same strategy for the next 125-day trading.
2. If normalised value is 1 and trader is using a strategy drawn from the pool,
   - Do not publish the strategy into the pool but update the strategy's score in the pool in the pool using their six-month ROP.
   - Use the strategy for the next 125-day trading.
3. If normalised value is less than 0.9, trader has two options:
   a- With 0.5 probability, replace the current strategy with a selected strategy from the pool. The roulette wheel selection is used to select the better trading strategy from the pool and use this copied strategy for the next six-month trading.
   b- Or, with 0.5 probability, discard the current strategy and select another set of indicators as inputs, build 10 new models and use these models for the next 125-day trading.
4. If normalised value is between 1 and 0.9,
   - The trader can use the same strategy for the next 125-day trading.

Algorithm 2.9 Social Learning (Su 2005).

Results show that trading strategies were successful when integrating individual and social learning. The trader could control the purchase-sell timing, hence build wealth quicker. The work of (Kendal and Su 2003) showed that individual learning helped traders to learn to trade while the search for better information and knowledge in the global space by traders was achieved through social learning.

Su continues her work in (Su 2005), which was published in (Kendal and Su 2004, 2007), where the integration of individual and social learning was implemented on an imperfect evolutionary market. In (Kendal and Su 2003),

the 20 market indicators were static during the trading period which, in fact, does not accurately reflect real life. Therefore, new indicators were introduced into the simulated stock market and the artificial traders learned how to use them (Kendal and Su 2004, 2007). They started with ten indicators in the central pool, with another ten indicators being gradually introduced into the simulated stock market. This model used the same mechanism of individual learning as in (Kendal and Su 2003); however some modifications were applied to the social learning algorithm as follows:

1. If the normalised value is 1 and trader is not using a strategy drawn from the pool,
   - Publish the strategy into the pool and use the same strategy for the next 125-day trading.

2. If the normalised value is 1 and trader is using a strategy drawn from the pool,
   - Do not publish the strategy into the pool but update the strategy's score in the pool.
   - Use the strategy for the next 125-day trading.

3. If the normalised value is less than 0.9, trader has two alternatives:
   a- Replace the current strategy with a selected strategy from the pool, or
   b- Discard the current strategy and select another set of indicators as inputs, build 10 new models and use these models for the next 125-day trading.

4. If the normalised value is between 1 and 0.9, the trader has two alternatives,
   - With 70% probabilities, the trader can use the same strategy for the next 125-day trading, or
   - With 30% probabilities, the trader can choose to use a new set of indicators.

Algorithm 2.10 Modified Social Learning (Kendal and Su 2007).

Three types of studies were carried out in (Kendal and Su 2004, 2007). The first was about the adaptability and creativity of environmental variables,

in the imperfect environment, to the new traders. As initial settings, the first 10 indicators were introduced to the imperfect evolutionary market. The remaining 10 indicators were inserted into the market at a frequency of two indicators per every 125-day trading. During social learning, poor traders, who opt to replace their models with a new indicator, will have a dual chance to copy from both the central pool as well as the newly injected indicators to the market. The results demonstrated that there has been poor performance of the traders in dynamic environment variables in comparison to the traders in (Kendal and Su 2003).

The second study was on individual learning. The purpose was to examine the time needed by the traders to learn, by individual learning, and the frequency at which social learning is should take place. It is worth mentioning that there are two types of individual learning. Fast individual learning (every 5 trading days) and slow individual learning (every 25 trading days). Similarly, there are two types of social learning. Fast and slow social learning, every 125 and 250 trading days, respectively. Results obtained were mixed with some experiments doing better when fast individual learning was used, and others being superior when slow individual learning was employed. Similar results were obtained in social learning. These findings conclude that the nature of the problem is to decide on good parameter settings. Also, it is likely that the dynamic individual and social learning would do better than fixed learning frequencies.

The third study was on social learning. In this study, social learning was investigated under different circumstances in the trading society. Four different experimental settings were run. These were:

- Social learning was turned off (individual learning only);
- Individual and social learning were turned on (similar to the experiment in Kendal and Su 2003);
- Individual and social learning were turned on, but the normalised values were in the range between 1 and the mean value of φ^3;
- Individual and social learning were turned on, but the normalised value was between 0.9 and the mean value of φ.

In conclusion, the work in (Su 2005) indicated that the integrated individual and social learning was of help in making successful trades in the stock market. Moreover, it showed that social learning led to superior traders.

2.8 *N*-TUPLE SYSTEMS

Work on optical character recognition, utilising *n*-tuples, can be dated back to the late 1950's (Bledsoe and Browning 1959). *N*-tuples operate by sampling *n* random points. If *m* is the number of possible values for each sample point, we can define an *n* digit number in base *m*, and use it as an index into a range of weights. *N*-tuples are in some ways similar to support vector machines (SVM), and is also related to Kanerva's sparse distributed memory model (Kanerva 1988). Figure 2.15 (Lucas 2008) shows an example

of a single 3-tuple, which is sampling 3 squares along an edge into the corner for the game of Othello, where each square of the game's board has three possible values (white=0, vacant=1, and black=3). In this case we will have 27 tuples (*m*=3, *n*=3).

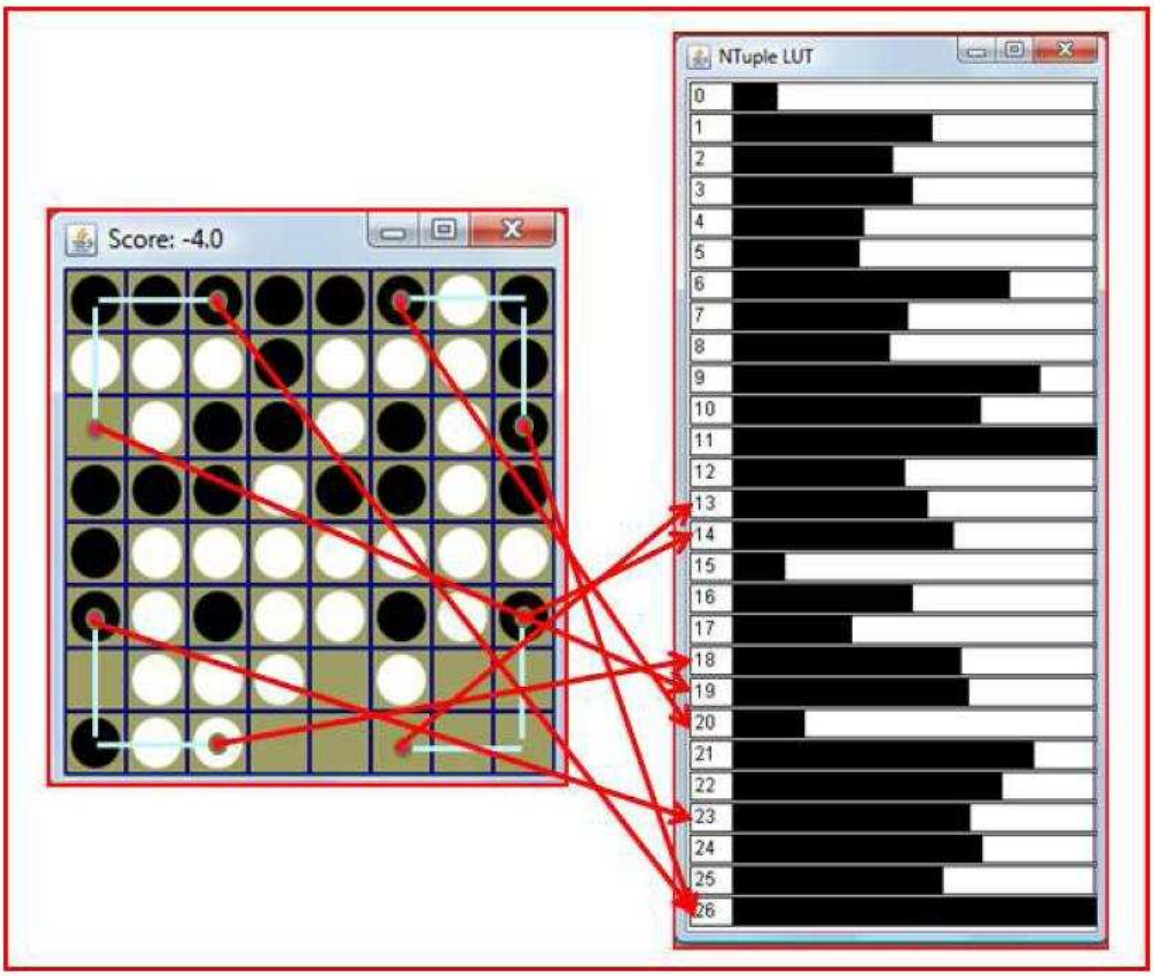

Figure 2.15 The system architecture of the N-Tuple-based value function, showing a single 3-tuple sampling at its eight equivalent positions, equivalent under reflection and rotation (Lucas 2008).

N-tuples indexing projects the low dimensional board into a high dimensional sample space. There are several varieties of *n*-tuple systems. The first model incorporates *n*-tuple systems into hardware, which is easy and effective given that indexing can be carried out so naturally in hardware. In its simplest form

a memory configuration with a single-bit width (a binary *n*-tuple) is used. Each memory location in a binary *n*-tuple records whether an address has occurred during training or not. As all addresses will eventually occur, excessive training can lead to poor performance, which is a distinct disadvantage of such systems. For this reason, later *n*-tuple systems tended to store continuous value weights, or probabilities. When trained on supervised data, probabilistic *n*-tuple systems can be trained using single-pass maximum likelihood techniques, where the probability of occurrence of each address is estimated as the number of happenings during training divided by the number of occurrences of all addresses in the *n*-tuple.

Although the basic idea of *n*-tuple systems is simple, obtaining good from them in practice is often difficult and the design may have to be carefully considered. There are many designs to draw inspiration from, including continuous *n*-tuples used for face recognition (Lucas 1998), scanning *n*-tuples for sequence recognition (Lucas and Amiri 1996), scanning *n*-tuple grid for OCR (Lucas and Cho 2005) and the *n*-tuple classifier (Rohwer and Morciniec 1998; Lucas 2003). Bit-plane decomposition methods have also produced interesting results (Hoque et. al. 2002). More recently, a back-propagation training rule based on optimising a cross-entropy measure was introduced by (Lucas 2003). For excellent introductions to standard *n*-tuple systems, please refer to (Ullman 1969; Rohwer and Morciniec 1996).

Lucas also introduced *n*-tuple systems as position value functions for the game of Othello (Lucas 2008). The *n*-tuple architecture is evaluated for use with temporal difference learning. Performance is compared with previously

developed weighted piece counters and multi-layer perceptrons. The *n*-tuple system is able to defeat the best performing of these after just five hundred games of self play learning. The conclusion is that *n*-tuple networks learn faster and are superior to other, more conventional, approaches. The success of applying *n*-tuple to the game of Othello inspired us to apply *n*-tuple to the game of checkers.

2.9 TEMPORAL DIFFERENCE LEARNING

Temporal Difference Learning (TDL) method (Sutton 1988) has been used to estimate the value of positions. This method can be defined as a Reinforcement Learning method driven by the difference between two consecutive state values aiming at adjusting former state values which minimise the difference between two successive state values. The multiplication of a learning parameter, α, by the sum of the temporal difference errors between two successive state values represents the change in the value of the state. These temporal differences are weighed exponentially according to the difference in time.

Sutton introduced TD (λ), which is used "to weight the influence of the current evaluation function value for weight updates of previous moves" (Sutton 1988). The λ term is decay-rate parameter. It determines the extent to which learning is affected by subsequent states. A λ of zero indicates learning only from the next state. A λ of one indicates learning only from the final reinforcement signal; in the case of the game playing, the final results (win, lose and draw).

In TDL the weights of the evaluation function are updated during game play using a gradient-descent method. Let x be the board observed by a player about to move, and similarly x' the board after the player has moved. Then the evaluation function may be updated during play using the following equation (Lucas and Runarsson 2006):-

$$w_i = w_i + \alpha[v(x') - v(x)](1 - v(x)^2)x_i \quad (2.11)$$

Where:

- $v(x) = \tanh(f(x)) = \frac{2}{1+\exp(-2f(x))} - 1$ is used to force the value function v to be in the range -1 to 1.
- w_i represents the weight to be updated.
- $f(x)$ represents the state of the board.

If x' is a terminal state then the game has ended and the following update is used:

$$w_i = w_i + \alpha[r - v(x)](1 - v(x)^2)x_i \quad (2.12)$$

Where r corresponds to the final utilities: +1 if the winner is Black, −1 when White, and 0 for a draw.

Temporal difference learning is a prediction-based method in which future behaviour is calculated using past experiences with a partly known system (Sutton 1988). Examples of temporal difference learning include Samuel's checkers program (Samuel 1959), and works on Adaptive Heuristic Critic (Barto et. al. 1983; Sutton 1984). An example of successful temporal difference learning in games is (Tesauro 2002), where Tesauro produced a strong backgammon program, TD-Gammon that is able to teach itself to play backgammon solely by playing against itself and learning from the results, starting from random initial play. Another example can be found in

(Runarsson and Lucas 2005), where temporal difference learning is used to evaluate the position on small-board Go (5x5 board). It was compared to a co-evolutionary approach. Temporal difference learning was shown to learn faster than a co-evolutionary approach, yet the latter played at a higher level than the temporal difference player. Lucas and Runarsson (2006) found that temporal difference learning learns much faster than co-evolution in the game of Othello, but that properly tuned co-evolution can learn better playing strategies.

One last example of using temporal difference learning can be found in (Burrow and Lucas 2009), where temporal difference learning was found to perform more reliably (with a tabular function approximator) than an evolutionary approach in Ms. Pac-Man.

2.10 SUMARRY

This chapter has provided an overview of various artificial intelligence researches, which includes the basic algorithms that can be used in computer games. Evolutionary computation algorithms have also been described in addition to artificial neural networks. Many computer games were presented, including a detailed description of the design of Blondie24. Individual and social learning, *n*-tuple systems and temporal difference learning were also presented as we utilise these methods to enhance evolutionary checkers methodologies. The next chapter will describe the evolutionary checkers preliminaries that will form the foundation for the rest of this thesis.

Chapter Three

Evolutionary Checkers Preliminaries

3.1 INTRODUCTION

In this chapter a description of the implementation of an evolutionary checkers player, C_0, is presented as it will be used as a test bed for all the proposed algorithmic developments in this research. The structure and architecture of C_0 is mainly based on those used to construct Blondie24. Two-move ballot is also presented, together with the standard rating formula as both will also be used to test the outcome of the methods that are used in this research.

This Chapter is structured as follows: Section 3.2 describes the implementation of C_0, while section 3.3 describes the two-move ballot that is used in the game of checkers. Section 3.4 describes the standard rating formula, which is used to rate the checkers players. A summary of the chapter is presented in section 3.5.

3.2 C_0

In order to investigate our proposed extensions and enhancements to an evolutionary checkers system we firstly implemented an evolutionary checkers program, which we will refer to as C_0 throughout this thesis, in order to provide a firm foundation for our research. Our implementation has the same structure and architecture that Fogel utilised in Blondie24, with the exception that the value of the King is fixed to 2. Intuitively, the King is more valuable than an ordinary piece, and this is a well known, even to novice players. So putting the value of the King as two (or any other value that is greater than an ordinary piece value) will not be considered as knowledge injection to the program. Algorithm 3.1 (Chellapilla and Fogel 1999, 2001) is used to construct C_0. It is worth mentioning that C_0 used depth first search to expand the search space to a four ply depth, while a ply depth of six is used in all algorithms comparisons.

1- Initialise a random population of 30 neural networks (strategies) P_i, i=1,…,30, sampled uniformly [-0.2,0.2] for the weights and biases.

2- Each strategy has an associated self-adaptive parameter vector s_i, i=1,…,30 initialised to 0.05.

3- Each neural network plays (as red) against five other neural networks selected randomly from the population.

4- For each game, each competing player receives a score of +1 for a win, 0 for draw and -2 for a loss.

5- Games are played until either one side wins, or until one hundred moves are made by both sides, in which case a draw was declared.

6- After completing all games, the 15 strategies that have the highest scores are selected as parents and retained for the next generation. Those parents are then mutated to create another 15 offspring using the following equations:

$$s_i(j) = s_i(j)\exp(\, tN_j(0,1)\,),\ j = 1, \ldots, N_w \qquad (3.1)$$
$$w_i(j) = w_i(j) + s_i(j)N_i(0,1),\ j = 1, \ldots, N_w \qquad (3.2)$$

where N_w is the number of weights and biases in the neural network (here this is 5046), $t = \frac{1}{\sqrt{2\times\sqrt{N_w}}}$ = 0.0839, and $N_j(0,1)$ is a standard Gaussian random variable resembled for

---

every j.

7- Repeat steps 3 to 6 for 840 generations (this number was an arbitrary choice in the implementation of Blondie24).

Algorithm 3.1 C_0 adapted from (Chellapilla and Fogel 1999, 2001).

We run the above algorithm for about 19 days (Fogel required about six months, but technology has moved on in the past ten years). All our experiments were run on the same computer (1.86 GHz Intel core2 processor and 2 GB Ram). For comparison, Fogel used a 400-MHz Pentium II processor.

3.3 TWO-MOVE BALLOT IN CHECKERS

When the world's best players play the game of checkers, it often ends in a draw. To overcome this, and make the games more competitive, the Two-Move Ballot is used.

This was introduced in the 1870s (see Schaeffer 2009). The first two moves (each side's first move) are randomly chosen. There are 49 possibilities to play in this way, but research showed that six possibilities should be excluded, either because they were certain losses for one side, or because they were, at least, regarded as excessively unbalanced. Figure 3.1 shows all the positions for a checkers board, while table 3.1 shows all the 49 possibilities, where the six excluded ones are highlighted in Bold.

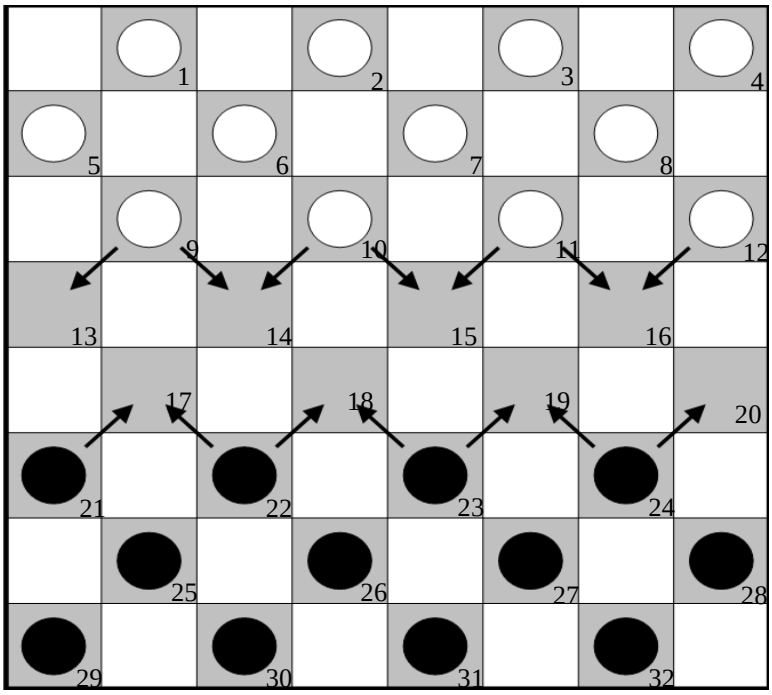

Figure 3.1 Checkers board with Black moves first.

No.	Two-move Ballot Possibility
1	21-17, 9-13
2	**21-17, 9-14**
3	**21-17, 10-14**
4	21-17, 10-15
5	21-17, 11-15
6	21-17, 11-16
7	21-17, 12-16
8	22-17, 9-13
9	22-17, 9-14
10	22-17, 10-14
11	22-17, 10-15
12	22-17, 11-15
13	22-17, 11-16
14	22-17, 12-16
15	22-18, 9-13
16	22-18, 9-14
17	22-18, 10-14
18	22-18, 10-15
19	22-18, 11-15
20	22-18, 11-16
21	22-18, 12-16
22	23-18, 9-13
23	**23-18, 9-14**
24	**23-18, 10-14**

25	23-18, 10-15
26	23-18, 11-15
27	23-18, 11-16
28	23-18, 12-16
29	23-19, 9-13
30	23-19, 9-14
31	23-19, 10-14
32	23-19, 10-15
33	23-19, 11-15
34	**23-19, 11-16**
35	**23-19, 12-16**
36	24-19, 9-13
37	24-19, 9-14
38	24-19, 10-14
39	24-19, 10-15
40	24-19, 11-15
41	24-19, 11-16
42	24-19, 12-16
43	24-20, 9-13
44	24-20, 9-14
45	24-20, 10-14
46	24-20, 10-15
47	24-20, 11-15
48	24-20, 11-16
49	24-20, 12-16

Table 3.1 The 49 possible two-move ballot openings.

Therefore, only 43, of the 49 available moves are considered. At the start of the game a card is randomly chosen indicating which of the 43 moves is to be played. The original game, with no forced opening moves, is called go-as-you-please (GAYP).

In order to make sure that the C_0 is not a 'fluke' of optimisation, we decided to construct ten players, comparing them using the idea of two move ballot and test if they are statistically the same by using student t-test (assuming unequal variances, $\alpha = 0.05$, and one-tail test) for the total number of wins and losses. The null hypothesis is that two players are the

same if the P value obtained from the *t*-test is greater than alpha. Table 3.2 shows the results.

	$C_0(1)$	$C_0(2)$	$C_0(3)$	$C_0(4)$	$C_0(5)$	$C_0(6)$	$C_0(7)$	$C_0(8)$	$C_0(9)$	$C_0(10)$	Σ Wins	Σ Loses
$C_0(1)$	-	22	25	20	19	17	23	24	22	20	192	200
$C_0(2)$	20	-	20	22	24	21	20	21	23	24	195	187
$C_0(3)$	21	19	-	20	21	19	18	23	21	22	184	192
$C_0(4)$	25	23	18	-	20	24	20	24	18	19	191	187
$C_0(5)$	24	20	23	19	-	18	20	22	22	21	189	188
$C_0(6)$	21	24	22	23	20	-	22	20	23	22	197	184
$C_0(7)$	24	18	20	21	21	19	-	19	18	20	180	180
$C_0(8)$	25	17	19	24	20	20	17	-	22	23	187	198
$C_0(9)$	21	22	21	18	22	24	20	24	-	22	194	193
$C_0(10)$	19	22	24	20	21	22	20	21	24	-	193	193

Table 3.2 Number of wins and losses (for the row player) out of 774 games.

Based on Table 3.1, there is no statistical difference between the players as the P value (**P-value=0.5**) for the one tail t-test is greater than alpha. So as all the players are statistically the same we decided to choose the player with the most number of wins to be our baseline player, C_0.

3.4 STANDARD RATING FORMULA

Checkers players are rated according to a standard system (following the tradition of the United States Chess Federation) where the initial rating for a player is R_0 = 1600 and the player's score is adjusted based on the outcome of a match and the rating of the opponent (Chellapilla and Fogel 2001):

$$R_{new} = R_{old} + C(Outcome - W) \qquad (3.3)$$

Where

- $W = 1/(1+10^{((Ropp-Rold)/400)})$
- *Outcome value is* 1 for Win, 0.5 for Draw, or 0 for Loss.
- R_{opp} is the opponent's rating.

- C = 32 for ratings less than 2100, C = 24 for ratings between 2100 and 2399, and C = 16 for ratings at or above 2400.

- R_{new} is the computed new rating based on an old rating of R_{old}.

It is clear that a player rating increases when a win occurs and decreases when a loss occurs, but the amount of increase or decrease depends on how big the difference between the rating of the player and its opponent's rating. It is also worth noting that constant factor C will be lower as the rating of the player increases, making it more difficult to gain or lose points. Standard designations for the level of play are shown in Table 3.3 (Chellapilla and Fogel 2001). While Table 3.4 shows some examples of using equation (3.3).

For the purpose of providing some form of statistical test, we will use 5000 different orderings for the 86 (each player plays 43 games as red and 43 games as white) games and then compute the mean and the standard deviation for the standard rating formulas. We say that a player is statistically better than his opponent if his mean value of the standard rating formula puts him in a level that is higher than his opponent. The determination of the player level is according to table 3.3. We note that the purpose of this paper is to compare the performance of the two players and not to measure their actual ratings, which could only realistically be done by playing against a number of different players.

Class	Rating
Senior Master	2400+
Master	2200-2399
Expert	2000-2199
Class A	1800-1999
Class B	1600-1799
Class C	1400-1599
Class D	1200-1399
Class E	1000-1199
Class F	800-999
Class G	600-799
Class H	400-599
Class I	200-399
Class J	below 200

Table 3.3 The relevant categories of player indicated by the corresponding range of rating score (Chellapilla and Fogel 2001).

R_{old}	R_{opp}	W	R_{new} (win)	R_{new} (draw)	R_{new} (lose)
1600	1600	0.5	1616	1600	1584
1600	1365	0.79	1606.57	1590.57	1574.57
1930	1600	0.87	1957.84	1918.16	1902.16
1750	2200	0.07	1779.77	1763.77	1747.77
2000	1400	0.97	2000.98	1984.98	1968.98
2100	2100	0.5	2112	2100	2088
1200	2000	0.01	1231.68	1215.68	1199.68

Table 3.4 Examples of Standard Rating Formula.

Analysing table 3.4, it is clear that the difference between the ratings for the players will proportionally affect the final rating for them. For example, suppose you have a player rated at 1200 (Class D) playing against a better opponent rated at 2000 (Expert). If the opponent wins then his rating will climb by less than a point and the rating for the player will decrease by the same amount, while if the player wins, his rating will increase by almost 32 points and the opponent rating will decrease by the same amount.

The Rating System is designed to make a smaller adjustment to a player rating once he reaches 2100 points and even smaller adjustment once reaching 2400 points. It gets very difficult to reach extremely high ratings as the player always needs to play and defeat the best players. For the highly rated players, there is no point playing against weaker players as an easy win wouldn't earn the master even one full rating point.

3.5 SUMMARY

This chapter has provided details of the implementation of C_0, which will be used as a test bed for the proposed methods that are used in subsequent chapters. The C_0 implementation was based on the same architecture and structure that was used for Blondie24. This chapter has also provided a description of the two moves ballot method and the standard rating formula that will be used in this thesis in order to compare the various enhancements that we propose. The introduction of a round robin tournament into an evolutionary checkers program is the first such enhancement and it will be presented in the next chapter.

Chapter Four

Introducing a Round Robin Tournament into Evolutionary Checkers

4.1 INTRODUCTION

In chapter three many preliminaries that will be used in the subsequent chapters in this thesis were presented. This chapter investigates the effects of introducing a round robin tournament into an evolutionary computer checkers system. Artificial neural networks, evolved via an evolution strategy, are utilised to evolve game playing strategies for the game of checkers by introducing a league structure into the learning phase of a system based on Blondie24. We believe that this will help eliminate some of the randomness in the evolution. Thirty feed forward neural network players are played against each other, using a round robin tournament structure, for 140 generations and the best player obtained is tested against an implementation of evolutionary checkers program (C_0). The best player will be tested against an online program, as well as two other strong programs.

This chapter has been structured as follows; Section 4.2 describes the experimental setup by showing the proposed algorithm in detail together with

a justification for the parameters choices. In section 4.3 the results for our experiments are presented, together with a discussion for those results. Finally, a summary for this chapter is presented in section 4.4. This chapter has been disseminated via the following publication: Al-Khateeb and Kendall (2009).

4.2 EXPERIMENTAL SETUP

In order to eliminate the randomness in the evolutionary phase of C_0 and hence produce a better player, a league competition between all the 30 neural networks is suggested, by making all the neural networks play against each other. This means that all networks would play, as a red player, against the other 29 players instead of only playing against five randomly chosen players, which was the case in Fogel's seminal work and our reimplementation, C_0. The total number of matches per generation in this model will be 870 (30 X 29) rather than 150 (30 X 5), as in the implementation of C_0. This increase in the number of matches will decrease the number of generations (140 verses 840) that can be played in the same amount of time, in order to provide a meaningful comparison against the original work, as C_0 has a total of 126,000 games (30 X 5 X 840) so Blondie24-RR (a player obtained as a result of applying the proposed algorithm) needs 140 generations to play a similar (actually slightly less) number of games (29 X 30 X 140=121,800).

The only difference with algorithm 3.1 are in steps 3 and 7 (see algorithm 4.1), where every network competes against every other for 140 generations. We refer to this player as Blondie24-RR.

1- Initialise a random population of 30 neural networks (strategies), $P_i=1,\ldots,30$, sampled uniformly [-0.2,0.2] for the weights and biases.

2- Each strategy has an associated self-adaptive parameter vector, $s_i=1,\ldots,30$ initialised to 0.05.

3- Use a round robin tournament to play each neural network (as red) against every other neural network.

4- For each game, each competing player receives a score of +1 for a win, 0 for draw and -2 for a loss.

5- Games are played until either one side won, or until one hundred moves have been made by both sides, in which case a draw was declared.

6- After completing all games, the 15 strategies that have the highest scores are selected as parents and retained for the next generation. Those parents are then mutated to create another 15 offspring using equations (3.1) and (3.2).

7- Repeat steps 3 to 6 for 140 generations.

Algorithm 4.1 Blondie24-RR.

It is worth mentioning that Blondie24-RR is a result of a single optimisation run, therefore there is a chance that Blondie24-RR is a 'fluke'. In order to make sure that this might not the case, we decided to play the top five players of the last generation of the EA for Blondie24-RR using the idea of the two-move ballot and using student t-test (assuming unequal variances, $\alpha = 0.05$, and one-tail test) to see if the players are the same or not. Table 4.1 shows the results.

	P1	P2	P3	P4	P5	Σ Wins	Σ Loses
P1	-	24	20	25	19	88	77
P2	20	-	21	24	20	85	88
P3	18	21	-	22	23	84	85
P4	19	23	23	-	18	83	91
P5	20	20	21	20	-	81	80

Table 4.1 Number of wins and losses (for the row player) out of 344 games.

Based on Table 4.1, there is no statistical difference between the players as the P value (**P-value=0.5**) for the one tail t-test, for the total number of wins and losses, is greater than alpha. So one can conclude that the results in table 4.1 are of comparable performance. This provides some indication about trusting the single optimiser for Blondie24-RR.

4.3 RESULTS

To guage the effect of introducing a round robin tournament we play C_0 against Blondie24-RR. Bearing in mind the fact that both players are end products, a win result for our modified player should be seen as a success. Also we play several matches against an online program, which can be found at http://www.darkfish.com/checkers/checkers.html, in addition to playing against two strong checkers programs (their implementation details are not available in the freeware versions). The first one called WinCheck3D, which was created in 2001 by Jean-Bernard Alemanni using the C++ programming language. WinCheck3D is considered as one of the strongest computer checkers programs that can play at a master level. The details for

WinCheck3D can be found at http://pagesperso-orange.fr/alemanni/. The second program called SXcheckers, which is produced by 504 software studio, is a strong checkers program with a strong AI component. SXcheckers can play at a human master level and has managed draw against WinCheck3D. The details for SXcheckers can be found at http://www.cs.504.com/checkers. The following subsections show the results.

4.3.1 Results When Playing Blondie24-RR Against C_0

In order to test the outcome of the proposed method, Blondie24-RR was set to play two matches (as red and as white) against C_0, Table 4.2 shows the results. It is worth mentioning that both players are biased (playing stronger games) towards playing as red.

	C_0 (red)	**C_0 (white)**
Blondie24-RR (red)	-	Win
Blondie24-RR (white)	Draw	-

Table 4.2 Blondie24-RR Against C_0.

Analysing the results in table 4.2, Blondie24-RR (after 140 generations) played two matches (one as red and one as white) against C_0. Blondie24-RR won as red (starts first) against C_0, the result was a draw when Blondie24-RR moves second. This clearly reflects a success for our hypothesis based on the

fact that both players are *end products.* It should be noted that both players will always play with the same strategy due to their deterministic nature.

4.3.2 Results When Playing Blondie24-RR Against Online Program

In order to test the outcome of the proposed method, C_0 and Blondie24-RR played two matches (as red and as white) against an online checkers program. Table 4.3 shows the results. It is worth mentioning that C_0 and Blondie24-RR are biased (playing stronger games) towards playing as red.

	Online (red)	**Online (white)**
C_0 (red)	-	Win (with four piece difference)
C_0 (white)	Win (with two piece difference)	-
Blondie24-RR (red)	-	Win (with seven piece difference)
Blondie24-RR (white)	Win (with four piece difference)	-

Table 4.3 C_0 and Blondie24-RR Against an Online Checkers Program.

The results in table 4.3 show that C_0 won as red (with a four piece advantage) and as white (with a two piece advantage) against this online program. The results in table 4.3 also show that Blondie24-RR won as red

(with a seven piece advantage) and as white (with a four piece advantage) against this online program. This reflects another success for our hypothesis as it is clear that Blondie24-RR performed better than C_0, with the piece advantage that each player gained supporting the conclusion.

4.3.3 Results When Playing Blondie24-RR Against WinCheck3D

Table 4.4 shows the results of playing C_0 and Blondie24-RR against WinCheck3D. In this case C_0 and Blondie24-RR were set to play two matches (as red and as white) against WinCheck3D.

	WinCheck3D (red)	**WinCheck3D (white)**
C_0 (red)	-	Lose (with seven piece difference)
C_0 (white)	Lose (with eight piece difference)	-
Blondie24-RR (red)	-	Lose (with two piece difference)
Blondie24-RR (white)	Lose (with four piece difference)	-

Table 4.4 C_0 and Blondie24-RR Against WinCheck3D.

The results in table 4.4 show that C_0 lost as red (with a seven piece difference) and as white (with an eight piece difference), while the results in table 4.4 also show that Blondie24-RR lost as red (with a two piece difference) and as white (with a four piece difference) against WinCheck3D. Several matches were played with WinCheck3D in order to investigate whether it is deterministic or not. The results were the same, indicating that the player always responds with the same moves. These results show that Blondie24-RR is performing better than C_0. Losing by two checkers is still a loss, but in this experiment we want to compare the performance of Blondie24-RR with C_0 and not with those computer programs, bearing in mind that all of them are end products.

4.3.4 Results When Playing Blondie24-RR Against SXcheckers

In order to further test the outcome of the proposed method, C_0 and Blondie24-RR were set to play two matches (as red and as white) against SXcheckers. Table 4.5 shows the results. It is worth mentioning that C_0 and Blondie24-RR are biased (playing stronger games) towards playing as red.

	SXcheckers (red)	**SXcheckers (white)**
C_0 (red)	-	Lose (with eight piece difference)
C_0 (white)	Lose (with eight piece difference)	-
Blondie24-RR (red)	-	Lose (with four piece difference)
Blondie24-RR (white)	Lose (with five piece difference)	-

Table 4.5 C_0 and Blondie24-RR Against SXcheckers.

The results in table 4.5 show that C_0 lost as red (with an eight piece difference) and as white (with an eight piece difference). The results in table 4.5 also show that Blondie24-RR lost as red (with a four piece difference) and as white (with a five piece difference) against SXcheckers. Several matches were played with SXcheckers in order to investigate whether it is deterministic or not. The results were the same, indicating that the player always respond with the same moves. These results show that Blondie24-RR is performing better than C_0. Losing by four checkers is still a loss, but in this experiment we want to compare the performance of Blondie24-RR with C_0 and not with those computer programs, bearing in mind that all of them are end products.

4.3.5 Results When Playing Blondie24-RR Against C_0 Using Two-Move Ballot.

When playing only two games between the players there is a possibility that we could just have well have found an unlucky flaw in one player, or the other. In order to avoid this we decided to compare the performance of Blondie24-RR over C_0 by using Two-Move Ballot. The results are shown in table 4.6 and figure 4.1.

	Opponent: C_0		
	Win	Draw	Lose
Blondie24-RR	47	26	13

Table 4.6 Blondie24-RR Against C_0 using the Two-Move Ballot.

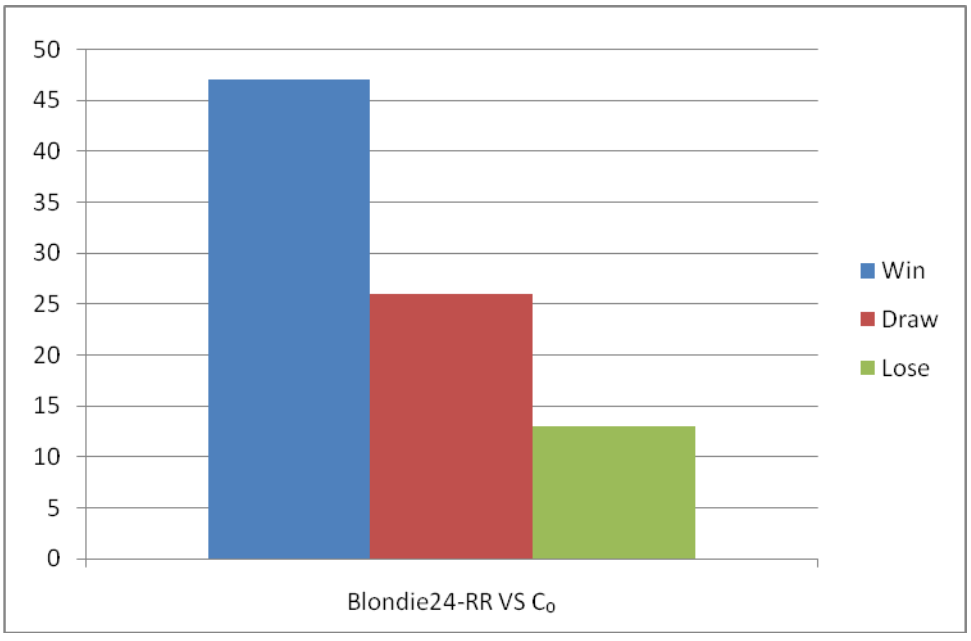

Figure 4.1 Results when Playing Blondie24-RR against C_0 using the Two-Move Ballot.

The results in table 4.6 show that Blondie-RR achieved 47 wins (from 86 games) over C_0, while C_0 only achieved 13 wins. There were 26 draws. It is clear that Blondie24-RR is superior to C_0. Table 4.7 shows the mean and the standard deviation of the players' ratings after 5000 different orderings for the 86 played games.

	Mean	**SD**	**Class**
Blondie24-RR	1251.67	25.76	D
C_0	1102.89	25.06	E

Table 4.7 Standard rating formula for Blondie24-RR and C_0 after 5000 orderings.

The results in table 4.7, obtained using 5000 different orderings for the 86 games (obtained using the two-move ballot) show that Blondie24-RR is better (using our definition given earlier with respect to players having a different rating class) than C_0, as the average ratings put Blondie24-RR in class D (rating = 1251) and put C_0 in Class E (rating = 1102). It is worth mentioning that these are not the actual ratings for the players, as the purpose here is to compare the performance of Blondie24-RR against C_0. By using the student t-test (assuming unequal variances, $\alpha = 0.05$, and one-tail test), the results show that Blondie24-RR and C_0 are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha.

Based on all results above, it would seem appropriate to use the league structure, instead of only choosing five random opponents to play against during the evolutionary phase.

4.4 SUMMARY

In this chapter evolutionary neural networks, evolved via an evolution strategy, are utilised to evolve game playing strategies for the game of

checkers by introducing a league structure into the learning phase of a system based on Blondie24. We believe that this helps eliminate some of the randomness in the evolution. Thirty feed forward neural network players are played against each other, using a round robin tournament structure, for 140 generations and the best player obtained is tested against C_0 (the evolutionary checkers program based on Blondie24). We also tested the best player against an online program, and Blondie24-RR was able to beat this program. Also we tested Blondie24-RR against two strong programs (WinCheck3D and SXcheckers). The results obtained are promising, although resulting in losses. The results showed that Blondie24-RR is better than C_0 by using two-move ballot and standard rating formula to test the outcome. Blondie24-RR was able to beat C_0 when all the pieces are in their original positions (i.e. without the two-move ballot).

Recent work for the superiority and progress in coevolution (Miconi 2009) showed that playing against a small number of opponents gives good results as long as it is the same set of individuals tested against all members of the population. This might/might not be the case for checkers, so further work need to be done to see if this is the case or not.

Now that we have shown that enhancements are possible to the evolutionary checkers, based on the Blondie24 framework, our future work will investigate if other changes are possible. We will investigate using individual and social learning methods and *n*-tuple systems in the next two chapters in order to further enhance the ability of C_0 and Blondie24-RR.

Chapter Five

Introducing Individual and Social Learning into Evolutionary Checkers

5.1 INTRODUCTION

Chapter Four investigated the effects of introducing a round robin tournament into an evolutionary computer checkers and eliminate some of the randomness in the evolution of an evolutionary checkers program based on the architecture of Blondie24. The motivation of the work in this chapter is inspired by the success of Blondie24 but we hypothesise that the introduction of an individual and social learning mechanism will evolve a superior player. The resulting player will be tested against C_0 and Blondie24-RR.

This chapter will also investigate including round robin into the individual and social learning algorithm. This is done by playing the resulting player against C_0, Blondie24-RR and against the player that will be obtained from introducing individual and social learning into evolutionary checkers.

This chapter has been structured as follows; Section 5.2 describes the individual and social learning mechanism. The experimental setup is described in section 5.3. In section 5.4 results are presented. Section 5.5

shows the result of introducing round robin into the individual and social learning algorithm, along with a discussion on those results. Finally a summary for this chapter is presented in section 5.6. This chapter has been disseminated via the following publication: Al-Khateeb and Kendall (2011a).

5.2 INDIVIDUAL AND SOCIAL LEARNING

Humans, when developing strategies to defeat other humans, use a variety of techniques. For example, humans can improve their strategy by themselves or through learning from the experience of competing with other humans. Developing their own strategies based on a copy of a better player model is another technique utilised by humans.

In other words, humans can learn through individual and social learning. According to (Simon 1997), "*learning from others*" is called social learning. In general, social learning can be defined as learning indirectly from the experiences of others (as opposed to one's own experiences). In competitive learning (Rosin and Belew 1997), in order to survive to the next generation, all the players will play against each other. The sources of inspiration for our work can be found in (Kendall and Su 2003, 2007), (Chen 2004), (Yamamoto 2005) and (Chen and Yeh 2001), where a simulated stock market used co-evolving neural networks (evolved through a process of individual and social learning) was used. Agent-based computational economics is by far the most common use of social learning research (Kendall and Su 2003) and (Vriend 2000). In individual learning, the agents learn solely from their own

experience while in social learning, it is the other agents' experience that form the source of learning for the agents (Vriend 2000).

In this work, individual and social learning are utilised in two stages. The player will accumulate experience and undertake individual learning by playing against five other players. After a certain time has elapsed we enter a social learning phase when players are able to learn from each other.

To further expand on the concept of individual and social learning, in an automated game playing context, individual learning is defined as a player which learns and generates a strategy by himself from the cumulative experience gained through playing against other players. The player neither opts to copy another strategy from other players nor replaces its own strategy with a new strategy. This is in contrast to the idea of social learning where the player is given the chance to copy or generate a new strategy to replace its current one. That is, the player has the option to evolve its own strategy through individual learning. However; if the strategy is not good enough, it has the option of either copying a better strategy from a pool of accumulated good strategies or creating a new random strategy.

Best strategies from the population are retained in a social pool. This pool is made available to those players which are not performing well. In this respect it closely resembles hall of fame (Rosin and Belew 1997), where the progress of learning is tested against a panel of all the best evolved players at every generation. There are two reasons to save the best players at every generation. Firstly is to contribute genetic material to future generations. Secondly is for the purpose of testing. Hall of fame has been applied to many

games such as Nim and 3-D Tic-Tac-Toe and has been shown to be successful (Rosin and Belew 1997).

In social learning the player has the opportunity to replace their existing strategy with another one selected from the social pool in the hope that the selected strategy is better than their current one. All strategies in the social pool have their own score, updated over time. Algorithm 5.1 shows the activities in social learning.

1. Rank the players in descending order.
2. Copy the best player or players (if more than one) to the social pool.
3. For the rest of the players, there are two possibilities,
   (a) If the player is satisfied with his current strategy (based on their current score), retain that strategy,
   (b) If the player is not satisfied with their current strategy, three alternatives are available,
      i. Copy a strategy from the pool;
      ii. Create a new random strategy;
      iii. Retain their current strategy.

Algorithm 5.1 Social Learning Activities.

When considering social learning, it is interesting to compare it with the island model in evolutionary computation. In an island model, each individual in a sub-population evolves independently (Spieth et. al. 2004). Moreover, the best player from a sub-population can migrate to another sub-population, if and only if it is the better strategy. However, there is no creation of a new strategy in the sub-population. In social learning, as mentioned above, the individual players have the opportunity to copy a better strategy, retain their current strategy or generate a new random strategy.

The individual and social learning mechanism that we utilise is also different to Case-Injected Genetic Algorithm (CIGAR) (Louis and Miles 2005) and (Miles et. al. 2004) that combines genetic algorithms with case-based reasoning to play a computer strategy game. CIGAR works by injecting the best strategies (players) obtained from past games into the future population of a genetic algorithm in order to try and produce better players. This can be done along with a suitable representation. Results demonstrate that case injection can produce superior players.

Cultural algorithms are also different to individual and social learning mechanisms since cultural algorithms (Reynolds 1979, 1994) are models of evolutionary learning that are set to emulate cultural evolutionary processes. Two levels of evolution constitute a cultural algorithm, namely, the microevolution in a population space and the macroevolution in a belief space. Utilising an acceptance function, the experiences of individuals in the population space are employed to create problem solving knowledge which is then stored in the belief space. The knowledge is manipulated by the belief space and this subsequently guides the evolution of the population space through an influence function. A fraud detection system was designed by (Sternberg and Reynolds 1997) who used a cultural algorithm-based evolutionary learning approach to learn about the behaviour of a commercial rule-based system for detecting fraud. The acquired knowledge in the belief space of the cultural algorithm is then used to re-engineer the fraud detection system. Another application of cultural algorithms is in modelling the evolution of complex social systems (Reynolds et. al. 2003, 2005). Furthermore, the application of cultural algorithms for function optimization

problems in dynamic environments has been described by (Reynolds and Saleem 2001, 2004) and (Reynolds and Peng 2004). In their experiments, the dynamic environment is modelled as a two-dimensional plane on which four cones of varying heights and slopes are haphazardly positioned. At certain generations, the four cones change their locations on the plane hence the location of the optimum solution is constantly changing. When applied to the problem of finding the new optima in dynamic environments, (Reynolds and Saleem 2001) demonstrated that the cultural algorithm is superior compared to an evolutionary algorithm with only a single-level evolution. (Reynolds and Peng 2004) discuss how the learning of knowledge in the belief space warrants the adaptability of cultural algorithms. (Reynolds and Saleem 2004) further examine the contributions of various types of knowledge from the belief space in piloting the quest for the best solutions in both deceptive and non-deceptive environments.

5.3 EXPERIMENTAL SETUP

Our hypothesis is that the introduction of social learning into an evolutionary checkers system will provide a richer environment for learning. The players outside the social pool are called individual players, all of which attempt to develop their own strategy. At certain times, the best players are drawn from the social pool to replace poorly performing individual players.

In our experiments, we have made some modifications to the algorithm described in (Kendall and Su 2007) in order to investigate how to increase

the number of players in the social pool, thus, producing a larger number of strategies that can be copied by individual players.

We propose two phases. The first will use individual learning, with the best players being copied to the social pool after every *M* generations. In the second phase social learning occurs every *N* generations. In comparison to (Kendall and Su 2007), we copy strategies to the social pool more often (they called a social learning phase at every generation for 30 generations). It is worth mentioning that there is no maximum size fir the social pool, as setting maximum pool size can limit the number of players to be copied into.

In fact a decision for the number of generations to be considered for the individual phase and the learning phase was taken after checking many values and the experiments showed that *M*=5 and *N*=10 were suitable. Algorithm 5.2 represents our experimental setup.

1- Initialise a random population of 30 neural networks (players) sampled uniformly [-0.2,0.2] for the weights.

2- Each player has its associated self-adaptive parameter, initialised to 0.05.

3- Initialise M (frequency of individual learning) and N (frequency of social learning).

4- For each player in the current population, randomly chose five players to play against.

5- For each game, the player receives a score of +1 for a win, 0 for draw and -2 for a loss.

6- Games are played until either side wins, or until one hundred moves are made by both sides, in which case a draw is declared.

7- If the generation number is exactly divisible by M and not by N then

- Select the best player(s) with the highest score (if two or more players have equal scores, we will select all those players) and copy them to the social pool.

- Select the best 15 players and mutate them to get 15 offspring using equations (3.1) and (3.2).

8- If the generation number is exactly divisible by N then for all players, i, do:
- Normalize the individual scores (values between 0 and 1) for all the players using the following equation:

$$V_i = \frac{(X_i - Min)}{(Max - Min)} \qquad (5.1)$$

where V_i is the normalized value for player i, Min and Max is the lowest and highest score in the current population among all players, X_i is the score of player i before being normalized.

- If the normalised value is 1 and the player is not using a strategy drawn from the pool, then publish the strategy into the pool.
- If the normalised value is 1 and the player is using a strategy drawn from the pool then do not publish the strategy into the pool but update the strategy's score in the pool.
- For the rest of the players, there are two cases:-
  1. If the normalised value is between 1 and 0.9, then the player is satisfied with his current strategy and retains it.
  2. If the normalised value is less than 0.9, then the player is not satisfied with his current strategy. The player has three options:-
     a- With 1/3 probability, replace the current strategy by copying a new strategy from the pool. Roulette wheel selection is used to select the new strategy from the pool.
     b- With 1/3 probability, replace the current strategy by creating a new random strategy.
     c- With 1/3 probability, retain the current strategy.

9- If the generation number is not exactly divisible by M or N then

- Select the 15 best players and mutate them to get 15 offspring using equations (3.1) and (3.2).

10- Repeat steps 4-9 for K generations or for specified time.

Algorithm 5.2 Individual and Social Learning.

Two experiments were carried out. The first determined the best values for the number of generations to determine where the individual (*M*) and social (*N*) phases occur. This experiment was also used to see the effects of increasing the number of players in the social pool. Different values for (*M*,*N*) were chosen, these being (100,200), (50,100), (20,50), (10,20) and (5,10), the players representing them were called:

1- C_{200} a player when *M*=100 and *N*=200.

2- C_{100} a player when *M*=50 and *N*=100.

3- C_{50} a player when *M*=20 and *N*=50.

4- C_{20} a player when M=10 and N=20.

5- C_{10} a player when M=5 and N=10.

In order to provide an additional comparison we also used a baseline player, C_1 (M=5, N=10), which took just the best player, line 7 in the algorithm, choosing randomly if there are more than one, and retained only this player in the social pool.

The second experiment uses the best player from the above experiments to investigate the effects of introducing individual and social learning for evolutionary checkers.

In order to provide a fair comparison, we run the above algorithms for 840 generations (126,000 games) that was required to produce C_0. All our experiments were run on the same computer (1.86 GHz Intel core2 processor and 2GB Ram).

Algorithm 5.2 presents three options for the player who is not satisfied with his current strategy. All of those options have an equal probability to occur, so there is no guarantee about which one of them makes the difference. Therefore table 5.1 shows a copy of the social pool after 160 generations to illustrate how the players learn from each others.

Player	Generation	Pool Score	Reused
1	100	734.22	0
2	110	734.22	1
3	120	734.22	1
4	130	1468.43	2
5	130	2202.65	4
6	140	2936.86	7
7	150	2936.86	5
8	160	3671.08	6
9	160	4405.29	7
10	160	4405.29	8

Table 5.1 Example of the Social Pool.

Table 5.1 clearly shows that individual and social learning provides learning to the evolved checkers program as many players in the social pool has been reused for at least one time. Another thing to notice is that the recent social pool players have a higher probability of being selected for copying by the individual players with a poor strategy.

5.4 RESULTS

To measure the effect of introducing individual and social learning into an evolutionary checkers system, a league structure between C_1, C_{200}, C_{100}, C_{50}, C_{20} and C_{10} was held, in order to determine the best values for *M* and *N*. Each player was set to play against all other players by using the two-move ballot. We play all of the 43 possible games, both as red and white, giving a total of 86 games. The games were played until either one side wins or a draw is declared after 100 moves for each player. The total number of games to be played is 430. Table 5.2 shows the results.

	C_1	C_{200}	C_{100}	C_{50}	C_{20}	C_{10}	Σ wins
C_1	-	22	14	12	10	8	66
C_{200}	35	-	29	22	16	10	112
C_{100}	39	25	-	21	17	12	114
C_{50}	40	37	26	-	21	18	142
C_{20}	47	41	32	27	-	15	162
C_{10}	59	55	49	41	34	-	238

Table 5.2 Number of wins (for the row player) out of 430 games.

It is worth mentioning that although each player is the result of a single run, the trends in performance are consistent. For example the wins vs. C_1, C_{200}, C_{100}, C_{50}, C_{20} and C_{10} are all increasing. i.e. although there is uncertainty in how representative each player is of the approach used to create it, the trends do suggest that the learning strategy used is more significant.

Based on the results in table 5.2, C_{10} received most wins, providing evidence that M=5, N=10 are the best values to use in the individual and social learning experiment. Also to support this conclusion, and to see the effects of introducing the individual and social learning to the game of checkers, we decided to play each player against C_0 and against Blondie24-RR, which is a result of our previous work to enhance Blondie24 obtained by introducing a round robin tournament into C_0 (see chapter four) by using two-move ballot. We play all of the 43 possible games, both as red and white, giving a total of 86 games. The games were played until either one side wins or a draw is declared after 100 moves for each player. The detailed results for each player {C_1, C_{200}, C_{100}, C_{50}, C_{20}, C_{10}} against both C_0 and Blondie24-RR are in tables 5.3 and 5.4 and in figures 5.1 and 5.2.

	Opponent:C_0		
	Win	Draw	Lose
C_1	20	22	44
C_{200}	27	31	28
C_{100}	30	30	26
C_{50}	40	21	25
C_{20}	44	22	20
C_{10}	51	20	15

Table 5.3 Results when Playing C_1, C_{200}, C_{100}, C_{50}, C_{20} and C_{10} against C_0 using the Two-Move Ballot.

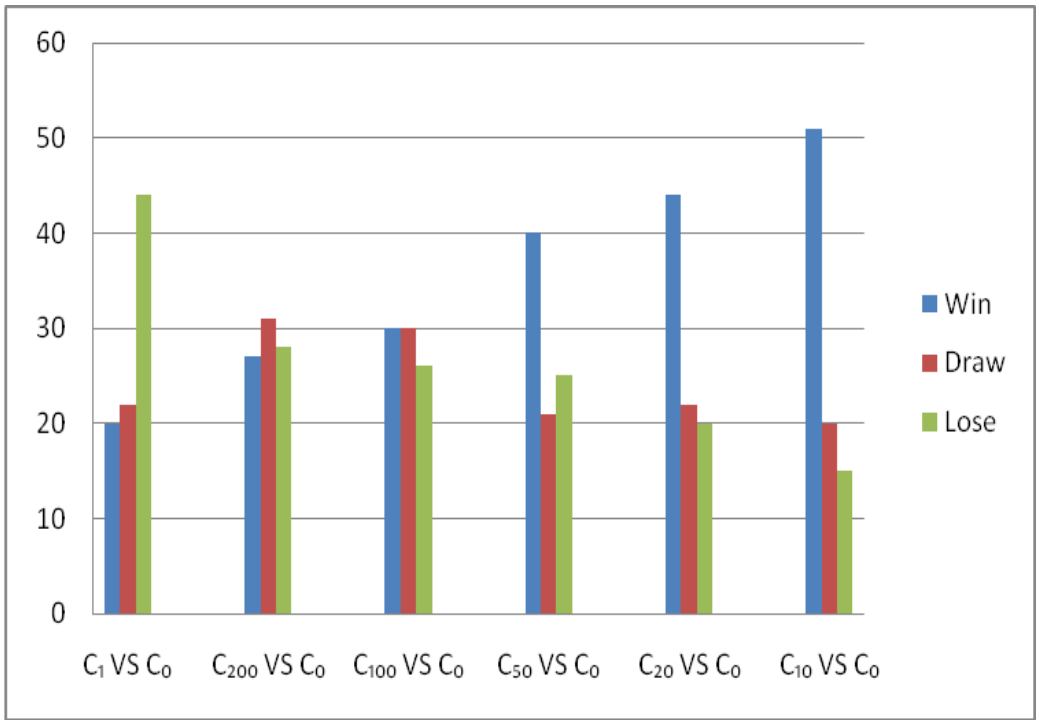

Figure 5.1 C_1, C_{200}, C_{100}, C_{50}, C_{20} and C_{10} against C_0.

	Opponent: Blondie24-RR		
	Win	Draw	Lose
C_1	17	16	53
C_{200}	20	29	37
C_{100}	22	28	36
C_{50}	30	17	39
C_{20}	31	25	30
C_{10}	43	18	25

Table 5.4 Results when Playing C_1, C_{200}, C_{100}, C_{50}, C_{20} and C_{10} against Blondie24-RR using the Two-Move Ballot.

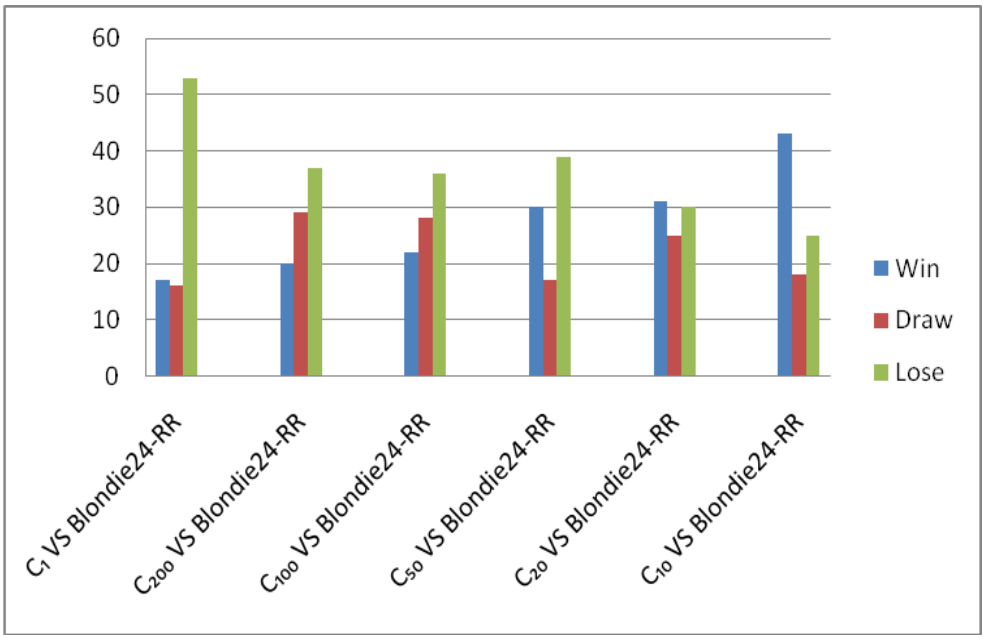

Figure 5.2 C_1, C_{200}, C_{100}, C_{50}, C_{20} and C_{10} against Blondie24-RR.

Table 5.5 summarises the results when playing against C_0 and against Blondie24-RR using a starting position where all pieces are in their original positions (i.e. no two-move ballot), while tables 5.6 and 5.7 show the mean and the standard deviation of the players' ratings after 5000 different ordering for the 86 played games.

		C_0	Blondie24-RR
C_1	Red	Lost	Lost
	White	Drawn	Lost
C_{200}	Red	Drawn	Lost
	White	Drawn	Lost
C_{100}	Red	Won	Lost
	White	Drawn	Lost
C_{50}	Red	Won	Lost
	White	Won	Drawn
C_{20}	Red	Won	Drawn
	White	Won	Drawn
C_{10}	Red	Won	Won
	White	Won	Won

Table 5.5 Summary of Wins/Loses when not Using Two-Move Ballot.

	Mean	SD	Class
C_1	1190.20	28.81	E
C_0	1288.07	27.47	D
C_{200}	1134.32	28.14	E
C_0	1148.69	26.87	E
C_{100}	1175.19	28.26	E
C_0	1173.69	27.01	E
C_{50}	1197.75	27.65	E
C_0	1110.59	26.62	E
C_{20}	1320.93	28.69	D
C_0	1227.47	27.63	D
C_{10}	1424.95	28.45	C
C_0	1288.49	27.49	D

Table 5.6 Standard rating formula for all the players against C_0 after 5000 orderings.

	Mean	SD	Class
C_1	1258.97	28.38	D
Blondie24-RR	1415.01	27.15	C
C_{200}	1190.51	26.64	E
Blondie24-RR	1258.98	25.38	D
C_{100}	1113.61	27.41	E
Blondie24-RR	1168.74	26.12	E
C_{50}	1303.45	30.02	D
Blondie24-RR	1339.21	28.67	D
C_{20}	1194.45	28.48	E
Blondie24-RR	1187.32	27.23	E
C_{10}	1205.58	25.88	D
Blondie24-RR	1082.35	25.07	E

Table 5.7 Standard rating formula for all the players against Blondie24-RR after 5000 orderings.

According to the results in tables 5.3, 5.4, 5.6 and 5.7, it is not recommended to use a social pool with only one player as both C_0 and Blondie24-RR is statistically better (using our definition given earlier with respect to players having a different rating class) than C_1, and by using the average value for the standard rating formula the results (when playing C_0

against C_1) put C_0 in class D (rating = 1288) and put C_1 in Class E (rating = 1190), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_0 and C_1 are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha. Also the results (when playing Blondie24-RR against C_1) put Blondie24-RR in class C (rating = 1415) and put C_1 in Class D (rating = 1258), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that Blondie24-RR and C_1 are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha.

The results in tables 5.3, 5.4, 5.6 and 5.7 also show there is no point of using the values (*M*=100 and *N*=200) for deciding where the individual and social learning phases occur as there is no statistical difference in the results in tables 5.3 and 5.6, as the results (when playing C_0 against C_{200}) put C_0 in class E (rating = 1148) and put C_{200} in class E (rating = 1134), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_0 and C_{200} are statistically the same as the P value (**P-value=0.5**) for the one tail t-test is greater than alpha. Also results in tables 5.4 and 5.7 showed that Blondie24-RR is better than C_{200}, as the results (when playing Blondie24-RR against C_{200}) put Blondie24-RR in class D (rating = 1258) and put C_{200} in class E (rating = 1190), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that Blondie24-RR and C_{200} are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha.

It is worth to mention that as C_{200} has a very few epochs of social learning, the performance results should be very similar to the C_0, which they are. This observation suggests lends evidence that the uncertainty in performance due to one run of the optimisation process is small.

Based on the results in tables 5.2 through 5.4, it is not sensible to use the values (*M*=50 and *N*=100) and (*M*=20 and *N*=50) for deciding where the individual and social learning phases occur. Although C_{100} and C_{50} look better than C_0, the results in table 5.6 put them in the same class, which means they are statistically the same, as the results (when playing C_0 against C_{100}) put C_0 in class E (rating = 1173) and put C_{100} in class E (rating = 1175), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_0 and C_{100} are statistically the same as the P value (**P-value=0.5**) for the one tail t-test is greater than alpha. The results (when playing C_0 against C_{50}) put C_0 in class E (rating = 1110) and put C_{50} in class E (rating = 1197), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_0 and C_{50} are statistically the same as the P value (**P-value=0.5**) for the one tail t-test is greater than alpha. Also Blondie24-RR, which is a result of a simple modification to the C_0, is better than C_{100} and C_{50} so it is not worth using the values (50 and 20) for *M* and the values (100 and 50) for *N*. The results in table 5.7 put Blondie24-RR in the same class as C_{100} and C_{50}, which means they are statistically the same, as the results (when playing Blondie24-RR against C_{100}) put Blondie24-RR in class E (rating = 1168) and put C_{100} in class E (rating = 1113), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_0 and C_{100} are statistically

the same as the P value (**P-value=0.5**) for the one tail t-test is greater than alpha. The results (when playing Blondie24-RR against C_{50}) put Blondie24-RR in class D (rating = 1339) and put C_{50} in class D (rating = 1303), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_0 and C_{50} are statistically the same as the P value (**P-value=0.5**) for the one tail t-test is greater than alpha.

The results in tables 5.3 show that using the values of (*M*=10 and *N*=20) for deciding where the individual and social learning phases occur, enhanced the process of C_0, but as there is not much difference in the results in table 5.4 as C_{20} is about equal to Blondie24-RR and the results in tables 5.6 and 5.7 showed that C_{20} is in the same class like C_0 and Blondie24-RR, as the results (when playing C_0 against C_{20}) put C_0 in class D (rating = 1227) and put C_{20} in class D (rating = 1320), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_0 and C_{20} are statistically the same as the P value (**P-value=0.5**) for the one tail t-test is greater than alpha. The results (when playing Blondie24-RR against C_{20}) put Blondie24-RR in class E (rating = 1187) and put C_{20} in class E (rating = 1194), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_0 and C_{20} are statistically the same as the P value (**P-value=0.5**) for the one tail t-test is greater than alpha. According to these results, it is not recommended to use the values 10 and 20 for *M* and *N*.

Based on the results obtained from tables 5.3 and 5.4 it is clear that increasing the number of players in the social pool will increase the

performance for the checkers player. Also the results in tables 5.6 and 5.7 showed that C_{10} is statistically better than both C_0 and Blondie24-RR, as the results (when playing C_0 against C_{10}) put C_0 in class D (rating = 1288) and put C_{10} in class C (rating = 1424), and by using student t-test (assuming unequal variances, $\alpha = 0.05$, and one-tail test), the results show that C_0 and C_{10} are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha. The results (when playing Blondie24-RR against C_{10}) put Blondie24-RR in class E (rating = 1082) and put C_{10} in class D (rating = 1205), and by using student t-test (assuming unequal variances, $\alpha = 0.05$, and one-tail test), the results show that C_0 and C_{20} are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha. Therefore it is recommended to use the values (M=5 and N=10) to determine where the individual and social learning phases occur.

In order to eliminate the randomness in choosing five random opponents (step 4 in algorithm 5.2) to play against in the evolutionary phase of C_{10} and hence produce a better player, a league competition between all the 30 neural networks is suggested, by making all the neural networks play against each other. This is based on the success of introducing round robin tournament into evolutionary checkers (see chapter four). The next section shows the results of introducing round robin into C_{10}.

5.5 Introducing Round Robin Tournament into C_{10}

The only difference between algorithm 5.2 and the algorithm to introduce the round robin tournament into C_{10} is in step 4, where every network competes against every other network using the same computer and for 140

generations. We refer to this player as C_{10}-RR. It is worth to mention that C_{10}-RR is constructed using the same values of *M* and *N* (*M*=5, *N*=10) that were used to construct C_{10}, i.e. both C_{10}-RR and C_{10} used 1/6 of their total number of generations.

In order to test the outcome of introducing round robin tournament into the evolutionary phase of C_{10}, C_{10}-RR is set to play against C_0, Blondie24-RR and C_{10} using two-move ballot. The results are shown in tables 5.8 through 5.10 and figure 5.3.

	Opponent:C_0		
	Win	Draw	Lose
C_{10}-RR	49	20	17

Table 5.8 Results when Playing C_{10}-RR against C_0 using the Two-Move Ballot.

	Opponent:Blondie24-RR		
	Win	Draw	Lose
C_{10}-RR	41	22	23

Table 5.9 Results when Playing C_{10}-RR against Blondie24-RR using the Two-Move Ballot.

	Opponent:C_{10}		
	Win	Draw	Lose
C_{10}-RR	35	23	28

Table 5.10 Results when Playing C_{10}-RR against C_{10} using the Two-Move Ballot.

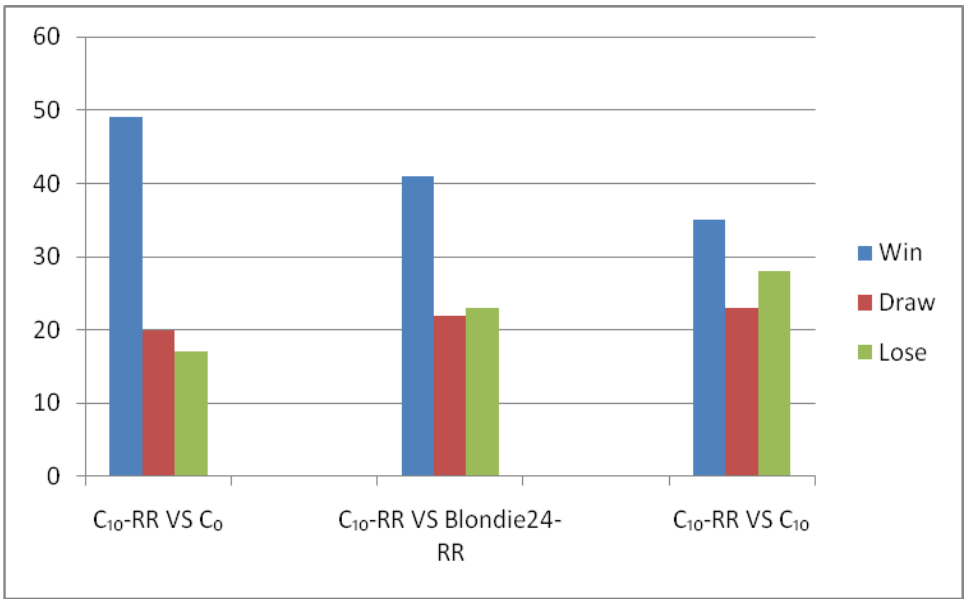

Figure 5.3 C_{10}-RR against C_0, Blondie24-RR and C_{10}.

Table 5.11 summarises the results when playing against C_0, Blondie24-RR and against C_{10} using a starting position where all pieces are in their original positions (i.e. no two-move ballot), while table 5.12 shows the mean and the standard deviation of the players' ratings after 5000 different ordering for the 86 played games.

		C_0	Blondie24-RR	C_{10}
C_{10}-RR	Red	Won	Won	Won
	White	Won	Won	Won

Table 5.11 Summary of Wins/Loses When not Using Two-Move Ballot.

	Mean	SD	Class
C_{10}-RR	1405.51	27.54	C
C_0	1264.71	26.66	D
C_{10}-RR	1250.44	28.71	D
Blondie24-RR	1171.91	27.61	E
C_{10}-RR	1229.99	29.08	D
C_{10}	1188.00	27.86	E

Table 5.12 Standard rating formula for playing C_{10}-RR against C_0, Blondie24-RR and against C_{10} after 5000 orderings.

The results in tables 5.8 and 5.12 show that C_{10}-RR is statistically better than C_0 as the results (when playing C_{10}-RR against C_0) put C_{10}-RR in class C (rating = 1405) and put C_0 in class D (rating = 1264), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_{10}-RR and C_0 are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha.

The results in tables 5.9 and 5.12 show that C_{10}-RR is statistically better than Blondie24-RR as the results (when playing C_{10}-RR against Blondie24-RR) put C_{10}-RR in class D (rating = 1250) and put Blondie24-RR in class E (rating = 1171), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_{10}-RR and C_0 are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha.

Finally the results in tables 5.10 and 5.12 show that C_{10}-RR is statistically better than C_{10} as the results (when playing C_{10}-RR against C_{10}) put C_{10}-RR in class D (rating = 1229) and put C_{10} in class E (rating = 1188), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_{10}-RR and C_0 are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha.

As C_{10}-RR is better than C_0 and Blondie24-RR and most importantly is better than C_{10}, then it seems quite appropriate to use the individual and social learning together with a round robin tournament in order to enhance the process of evolutionary checkers.

5.6 SUMMARY

This Chapter has introduced individual and social learning into an evolutionary checkers algorithm that is based on the Blondie24 architecture. The proposed algorithm shows promising results when tested against an implementation of an evolutionary checkers program, C_0, and also against a player obtained as a result of the previous efforts to introduce a round robin tournament into C_0.

Six players were implemented in order to see the effects of increasing the number of players in the social pool. Each player was implemented using a selected pair of values for the individual and social learning phases.

Based on the results in table 5.2 we can conclude that increasing the number of players in the social pool will increase the performance of the player. A value of 5 is the best to determine where the individual learning phase occurs. The value of 10 was found to be best when deciding where social learning should occur. Using these values C_{10} is the best player we obtained and is superior to C_0 (see tables 5.3, 5.5 and 5.6). Also the result in tables 5.4, 5.5 and 5.6 showed that C_{10} is better than Blondie24-RR.

Based on the results it would seem appropriate to use individual and social learning to enhance the evolutionary checkers systems.

Following the success of introducing round robin into evolutionary checkers in chapter four, we decided to use round robin within the individual and social learning framework. The resultant algorithm showed promising results as the best player, C_{10}-RR, was able to beat C_0, Blondie24-RR and C_{10}. We conclude

that it is appropriate to use a combination of round robin and individual and social learning in evolutionary checkers.

The next chapter will investigate if other enhancements to evolutionary checkers are possible by introducing an *n*-tuple architecture into evolutionary checkers and investigate the effect.

Chapter Six

Introducing *N*-tuple Systems into Evolutionary Checkers

6.1 INTRODUCTION

Chapter five showed that using individual and social learning for evolutionary checkers produced a superior player. This chapter investigates the effects of introducing *n*-tuple architecture into evolutionary computer checkers. Evolutionary neural networks, evolved via an evolution strategy, are utilised to evolve game playing strategies for the game of checkers. This will be done by introducing 5-tuple with random walk and 1-tuple to the learning phase of a system based on Blondie24 and also into a checkers program, which uses temporal difference learning (TDL). We believe that this helps in evolving a good player in a time that is faster than that required to evolve C_0, Blondie24-RR, C_{10} and C_{10}-RR. The resulting players will be tested against our baseline player, C_0, our round robin player (Blondie24-RR) and the two players from our individual and social learning experiments (C_{10} and C_{10}-RR).

This chapter is structured as follows; Section 6.2 describes the application of *n*-tuple to checkers and how the *n*-tuple framework is organised. Sections 6.3 and 6.4 describe the experimental setup and the results of using 5-tuple with a random walk. In sections 6.5 and 6.6 we describe the experimental setup and the results of using a 1-tuple. Sections 6.7 and 6.8 describe the experimental setup and the results of using 5-tuple with random walk utilises TDL. Sections 6.9 and 6.10 describe the experimental setup and the results of using 1-tuple with TDL. The comparison of 5-tuples with random walk and 1-tuple in an evolutionary checkers with TDL are presented in section 6.11. Finally a summary for this chapter is presented in section 6.12. This chapter has been disseminated via the following publication: Al-Khateeb and Kendall (2011b).

6.2 APPLICATION of *N*-tuple to EVOLUTIONARY CHECKERS

To apply an *n*-tuple system to Checkers, we firstly decide to cover all the 32 squares on the checkers board. The value function for the board is then calculated by summing over all table values indexed by all the *n*-tuples. Each *n*-tuple specifies a set of n board locations. Each *n*-tuple has an associated look-up table (LUT). The output for each *n*-tuple is calculated by summing the LUT values indexed by each of its equivalent sample positions. Each sample position is simply interpreted as an *n* digit quinary (base 5) number, since each square has five possible values (ordinary white, white king, vacant, ordinary black or black king). The board digit values were chosen as (vacant=0, ordinary white=1, white king=2, ordinary red=3, red king=4). The value function for a board is simply the sum of the values for each *n*-

tuple. Also the value of the piece difference for the checkers board is also added to the summation. For convenient training with error back-propagation the total output is passed through a tanh function.

The *n* positions can be arranged in a square, in a rectangle, or as random points scattered over the board. The results in this chapter are based on two types of sampling.

The first one is based on random walks, where each *n*-tuple is constructed by starting with each 32 squares on the board, and taking a random walk from that point. At each step of the walk, the next square is chosen as one of the immediate neighbours of the current square, which represents a legal checkers move. Each walk is for five steps. Each randomly constructed *n*-tuple had 5 sample points. The results in this chapter are based on 32 such *n*-tuples. One would expect some n-tuples to be more useful than others, and there should be scope for evolving the n-tuples sample points while training the look-up table values. A randomly constructed *n*-tuple sample is shown in Table 6.1 in which the samplings are based on the checkers board in figure 3.1. The experimental setup and its related results are shown in sections 6.3 and 6.4.

The second type of sampling is based on just one sample, which can be done by considering each square on the checkers board (see figure 3.1) at a time. In this case we will have 32 (as checkers board played on 32 squares) 1-tuple samples and the experimental setup and its results are shown in sections 6.5 and 6.6.

No.	5-tuple Sample
1	1,6,9,14,17
2	2,6,10,15,19
3	3,7,2,6,1
4	4,8,11,7,10
5	5,1,6,2,7
6	6,2,7,3,8
7	7,2,6,10,15
8	8,12,16,19,24
9	9,14,18,22,26
10	10,14,17,21,25
11	11,15,18,23,26
12	12,8,3,7,11
13	13,17,22,25,21
14	14,17,21,25,29
15	15,18,22,26,30
16	16,12,8,3,7
17	17,14,9,5,1
18	18,14,9,6,1
19	19,24,27,32,28
20	20,16,12,8,4
21	21,25,22,17,14
22	22,17,13,9,5
23	23,18,14,9,5
24	24,28,32,27,31
25	25,30,26,31,27
26	26,22,17,14,9
27	27,23,18,14,9
28	28,32,27,24,20
29	29,25,22,26,30
30	30,26,22,17,14
31	31,26,23,18,22
32	32,27,31,26,30

Table 6.1 The 32 random possible 5-tuple.

6.3 EXPERIMENTAL SETUP FOR 5-TUPLE WITH RANDOM WALK

Our hypothesis is that using *n*-tuple architecture will facilitate faster learning for the game of checkers and produce a better player.

The value function for the proposed *n*-tuple system is calculated by summing over all table values indexed by all the *n*-tuples. Algorithm 6.1 shows our *n*-tuple framework.

1- Take all the 32 possible checkers board squares. The n (n=5 for our experiments) positions can be arranged as random points scattered over the board. Each n-tuple is constructed by choosing each square on the board, and taking a random walk from that point. At each step of the walk, the next square is chosen as one of the immediate neighbours of the current square, which represents a legal checkers move.

2- There is a one Look-Up Table (LUT) for each 5-Tuple.

3- Since we have 5 types of pieces (our checker, our king, opponent's checker, opponent's king, and empty square), we require 5^5=3,125 possibilities for each n-tuple.

4- The values for the pieces will be:-

- 0 for opponent's checker.
- 1 for opponent's king.
- 2 for Empty Square.
- 3 for our checker.
- 4 for our king.

5- Initialise a population of 30 n-tuple networks (players), each one with total number of weights (32*3125)=100,000, are initialised to zero.

6- The result of evaluation the checkers board can be achieved by summing up all the corresponding LUT entries that are indexed by each n-tuple (in our case it will be only 32 entries each time).

7- Each n-tuple network plays against five other neural networks selected randomly from the population.

8- For each game, each competing player receives a score of +1 for win, 0 for draw and -2 for a loss.

9- Games are played until either one side wins, or until one hundred moves are made by both sides, in which case a draw is declared.

10- After completing all games, the 15 players that have the highest scores are selected as parents and retained for the next generation. Those parents are then mutated to create another 15 offspring by using the following equation:

$$w_i(j) = w_i(j) + N_i(0,1),\ j = 1, ..., N_w \qquad (6.1)$$

where N_w is the number of weights in the neural network and $N_i(0,1)$ is a standard Gaussian random variable resembled for every j.

11- Repeat the process for G generations.

Algorithm 6.1 5-tuple with random walk for evolutionary checkers.

It is worth mentioning that the number of weights for constructing the 5-tuple player (step 5 in the algorithm) is much bigger than those required to construct C_0 which increases the complexity of evolving the ANN player. Two 5-tuples would provide a chromosome with 6250 weights. But in this case not all the board squares covered, and this will prevent the ANN discovering many useful features in the board. For this reason we decided to use the 32 5-tuple system.

In order to provide a fair comparison, we run the above algorithm for the same number of generations (840 generations with 126,000 games) that was required to produce C_0. All our experiments were run on the same computer (1.86 GHz Intel core2 processor and 2GB Ram).

6.4 RESULTS FOR 5-TUPLE WITH RANDOM WALK

In order to test the effectiveness of algorithm 6.1, the best player (named C_5-tuple) was played against C_0, Blondie24-RR, C_{10}, and C_{10}-RR using two-move ballot. The detailed results are in table 6.2 and figure 6.1.

	Opponent: C_5-tuple		
	Win	Draw	Lose
C_0	43	23	20
Blondie24-RR	47	21	18
C_{10}	43	23	20
C_{10}-RR	58	15	13

Table 6.2 Results when Playing C_0, Blondie24-RR, C_{10} and C_{10}-RR against C_5-tuple using the Two-Move Ballot.

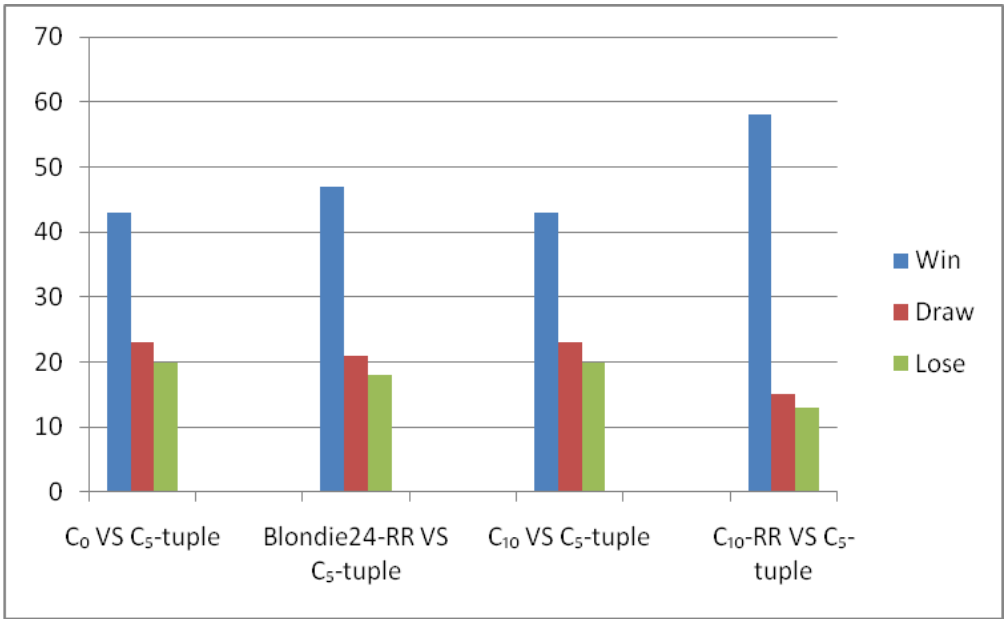

Figure 6.1 C_0, Blondie24-RR, C_{10} and C_{10}-RR against C_5-tuple using the Two-Move Ballot.

Table 6.3 summarises the results when playing against C_0, Blondie24-RR, C_{10} and against C_{10}-RR using a starting position where all pieces are in their original positions (i.e. no two-move ballot), while table 6.4 shows the mean and the standard deviation of the players' ratings after 5000 different ordering for the 86 played games.

		C_0	Blondie24-RR	C_{10}	C_{10}-RR
C_5-tuple	Red	Lost	Lost	Lost	Lost
	White	Drawn	Lost	Drawn	Lost

Table 6.3 Summary of Wins/Loses When not Using Two-Move Ballot.

	Mean	SD	Class
C_5-tuple	1175.50	27.06	E
C_0	1275.01	28.06	D
C_5-tuple	1195.76	26.94	E
Blondie24-RR	1321.18	27.86	D
C_5-tuple	1176.03	27.04	E
C_{10}	1274.40	28.05	D
C_5-tuple	1254.55	26.16	D
C_{10}-RR	1461.32	26.83	C

Table 6.4 Standard rating formula for C_5-tuple against C_0, Blondie24-RR, C_{10} and C_{10}-RR after 5000 ordering.

The results in tables 6.2 and 6.4 show that C_0 is statistically better than C_5-tuple as the results (when playing C_0 against C_5-tuple) put C_0 in class D (rating = 1275) and put C_5-tuple in class E (rating = 1175), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_0 and C_5-tuple are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha.

The results in tables 6.2 and 6.4 also show that Blondie24-RR is statistically better than C_5-tuple as the results (when playing Blondie24-RR against C_5-tuple) put Blondie24-RR in class D (rating = 1321) and put C_5-tuple in class E (rating = 1195), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that Blondie24-RR and C_5-tuple are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha.

The results in tables 6.2 and 6.4 show that C_{10} is statistically better than C_5-tuple as the results (when playing C_{10} against C_5-tuple) put C_{10} in class D (rating = 1274) and put C_5-tuple in class E (rating = 1176), and by using

student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_{10} and C_5-tuple are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha.

Finally the results in tables 6.2 and 6.4 show that C_{10}-RR is statistically better than C_5-tuple as the results (when playing C_{10}-RR against C_5-tuple) put C_{10}-RR in class C (rating = 1461) and put C_5-tuple in class D (rating = 1254), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_{10}-RR and C_5-tuple are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha.

The results clearly showed that C_5-tuple is not a good player compared with C_0, Blondie24-RR, C_{10} and C_{10}-RR. This could be because of the choice of the *n*-tuple sampling. In order to investigate this we use 1-tuple within algorithm 6.1. The experimental setup and the results are shown in sections 6.5 and 6.6.

6.5 EXPERIMENTAL SETUP FOR 1-TUPLE

This section describes how we applied 1-tuple architecture. The difference with algorithm (6.1) are in steps 1, 3 and 5 (see algorithm 6.2), where a 5-tuple will be replaced by 1-tuple. As a result the number of weights will be changed. We refer to this player as C_1-tuple.

1- **Take all the 32 possible checkers board squares. The n (n=1 for our experiments) positions can be arranged by choosing each square at a time.**

2- There is a one Look-Up Table (LUT) for each 5-Tuple.

3- **Since we have 5 types of pieces (our checker, our king, opponent's checker, opponent's king,**

and empty square), we require 5^1=5 possibilities for each n-tuple.

4- The values for the pieces will be:-

- 0 for opponent's checker.
- 1 for opponent's king.
- 2 for Empty Square.
- 3 for our checker.
- 4 for our king.

5- **Initialise a population of 30 n-tuple networks (players), each one with total number of weights (32*5)=160 are initialised to zero.**

6- The result of evaluation the checkers board can be achieved by summing up all the corresponding LUT entries that are indexed by each n-tuple (in our case it will be only 32 entries each time).

7- Each n-tuple network plays against five other neural networks selected randomly from the population.

8- For each game, each competing player receives a score of +1 for win, 0 for draw and -2 for a loss.

9- Games are played until either one side wins, or until one hundred moves are made by both sides, in which case a draw is declared.

10- After completing all games, the 15 players that have the highest scores are selected as parents and retained for the next generation. Those parents are then muted to create another 15 offspring by using equation (6.1).

11- Repeat the process for G generations.

Algorithm 6.2 1-tuple for evolutionary checkers.

It is clear from step 1 that the 1-tuple system is effectively a weighted piece counter as each square of the 32 squares is assigned its own tuple.

In order to provide a fair comparison, we run the above algorithm for the same number of generations (840 generations with 126,000 games) that was required to produce C_0. All our experiments were run on the same computer (1.86 GHz Intel core2 processor and 2GB Ram).

6.6 RESULTS FOR 1-TUPLE

In order to test the outcome of algorithm 6.2, the best player (named C_1-tuple) was played against C_0, Blondie24-RR, C_{10}, and C_{10}-RR using two-move ballot. The detailed results are in table 6.5 and figure 6.2.

	Opponent: C_1-tuple		
	Win	Draw	Lose
C_0	46	22	18
Blondie24-RR	48	21	17
C_{10}	47	24	15
C_{10}-RR	58	16	12

Table 6.5 Results when Playing C_0, Blondie24-RR, C_{10} and C_{10}-RR against C_1-tuple using the Two-Move Ballot.

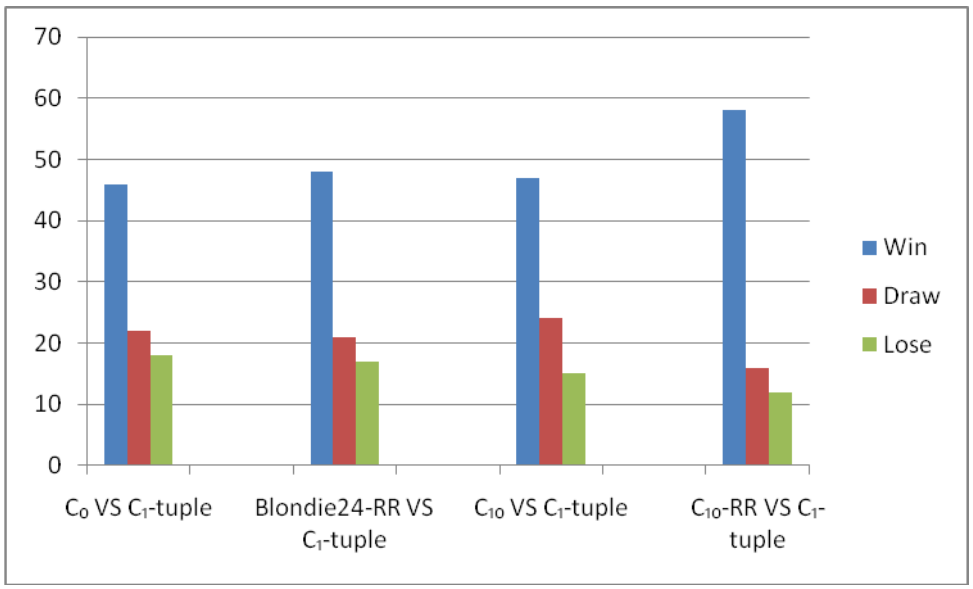

Figure 6.2 C_0, Blondie24-RR, C_{10} and C_{10}-RR against C_5-tuple using the Two-Move Ballot.

Table 6.6 summarises the results when playing against C_0, Blondie24-RR, C_{10} and against C_{10}-RR using a starting position where all pieces are in their original positions (i.e. no two-move ballot), while table 6.7 shows the mean

and the standard deviation of the players' ratings after 5000 different ordering for the 86 played games.

		C_0	Blondie24-RR	C_{10}	C_{10}-RR
C_1-tuple	**Red**	Lost	Lost	Lost	Lost
	White	Drawn	Lost	Drawn	Lost

Table 6.6 Summary of Wins/Loses When not Using Two-Move Ballot.

	Mean	SD	Class
C_1-tuple	1180.63	26.61	E
C_0	1303.47	27.53	D
C_1-tuple	1190.59	26.56	E
Blondie24-RR	1326.75	27.43	D
C_1-tuple	1138.90	25.74	E
C_{10}	1280.62	26.55	D
C_1-tuple	1235.57	25.86	D
C_{10}-RR	1447.89	26.49	C

Table 6.7 Standard rating formula for C_1-tuple against C_0, Blondie24-RR, C_{10} and C_{10}-RR after 5000 orderings.

The results in tables 6.5 and 6.7 show that C_0 is statistically better than C_1-tuple as the results (when playing C_0 against C_1-tuple) put C_0 in class D (rating = 1303) and put C_1-tuple in class E (rating = 1180), and by using student t-test (assuming unequal variances, $\alpha = 0.05$, and one-tail test), the results show that C_0 and C_1-tuple are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha.

The results in tables 6.5 and 6.7 also show that Blondie24-RR is statistically better than C_1-tuple as the results (when playing Blondie24-RR against C_1-tuple) put Blondie24-RR in class D (rating = 1326) and put C_1-tuple in class E (rating = 1190), and by using student t-test (assuming

unequal variances, α = 0.05, and one-tail test), the results show that Blondie24-RR and C_1-tuple are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha.

The results in tables 6.5 and 6.7 show that C_{10} is statistically better than C_1-tuple as the results (when playing C_{10} against C_1-tuple) put C_{10} in class D (rating = 1280) and put C_1-tuple in class E (rating = 1138), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_{10} and C_1-tuple are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha.

Finally the results in tables 6.5 and 6.7 show that C_{10}-RR is statistically better than C_1-tuple as the results (when playing C_{10}-RR against C_1-tuple) put C_{10}-RR in class C (rating = 1447) and put C_1-tuple in class D (rating = 1235), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_{10}-RR and C_1-tuple are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha.

The results clearly showed that C_1-tuple is not a good player compared with C_0, Blondie24-RR, C_{10} and C_{10}-RR. By considering these results together with those in section 6.4, we arrive at two conclusions. The first one is that the *n*-tuple architecture is not suitable for use within an evolutionary checkers system. The second conclusion is that 5-tuple constructed with random walk is slightly better than 1-tuple and this is because some of the tuples are like the 3X3 subsections that are used in the Blondie24 architecture.

In order to test whether the *n*-tuple architecture can be used with other evolutionary methods, we decided to use *n*-tuple with temporal difference learning, TD(0). The success of applying *n*-tuple to the game of Othello, together with TDL (Lucas 2008) inspired us to try the same approach within evolutionary checkers. Two experimental setups are considered. Firstly, one with the use of 5-tuple with random walks where the experimental setup and its related results are reported in sections 6.7 and 6.8. The second setup is with the use of a 1-tuple, where the experimental setup and its related results are reported in sections 6.9 and 6.10.

6.7 EXPERIMENTAL SETUP FOR 5-TUPLE WITH RANDOM WALK and TDL

The value functions for the proposed *n*-tuple system are calculated by summing over all table values indexed by all the *n*-tuples. Algorithm 6.3 presents our first experiment.

1- Take all the 32 possible checkers board squares. The n (n=5 for our experiments) positions can be arranged as random points scattered over the board. Each n-tuple is constructed by choosing each square on the board, and taking a random walk from that point. At each step of the walk, the next square is chosen as one of the immediate neighbours of the current square, which represents a legal checkers move.

2- There is a one Look-Up Table (LUT) for each 5-Tuple.

3- Since we have 5 types of pieces (our checker, our king, opponent's checker, opponent's king, and empty square), we require 5^5=3,125 possibilities for each n-tuple.

4- The values for the pieces will be:-

- 0 for opponent's checker.
- 1 for opponent's king.
- 2 for Empty Square.
- 3 for our checker.

- 4 for our king.

5- The total number of weights (32*3125)=100,000, are initialized to zero.

6- The result of evaluation the checkers board can be achieved by summing up all the corresponding LUT entries that are indexed by each n-tuple (in our case it will be only 32 entries each time).

7- In TDL the weights of the evaluation function are updated during game play using a gradient-descent method. Let x be the board observed by a player about to move, and similarly x′ the board after the player has moved. Then the evaluation function may be updated during play using the following equation (taken from Lucas and Runarsson 2006):-

$$w_i = w_i + \alpha[v(x') - v(x)](1 - v(x)^2) \quad (6.2)$$

Where

$v(x) = \tanh(f(x)) = \frac{2}{1+\exp(-2f(x))} - 1$ is used to force the value function v to be in the range -1 to 1.

8- If x' is a terminal state then the game has ended and the following update is used:

$$w_i = w_i + \alpha[r - v(x)](1 - v(x)^2) \quad (6.3)$$

Where r corresponds to the final utilities: +1 if the winner is Black, −1 when White, and 0 for a draw. Draw is declared after 100 moves by each player.

9- Repeat the process for G generations.

Algorithm 6.3 5-tuple with random walk for evolutionary checkers with TDL.

Three experiments were carried out, each one with different value for α (0.01, 0.001 and 0.001). Three players are constructed; those players are C_5-N0.01, C_5-N0.001 and C_5-N0.0001, where each player represents one of the selected values for α.

We run the above algorithm for the same number of games (126,000, which requires about two days), required to produce C_0 (which took 19 days to evolve). All our experiments were run on the same computer (1.86 GHz Intel core2 processor and 2GB Ram).

6.8 RESULTS FOR 5-TUPLE WITH RANDOM WALK AND TDL

In order to determine which value is suitable for α from the three selected values, C_5-N0.001, C_5- N0.01 and C_5-N0.0001 were played against each other by using the idea of a two-move ballot. Table 6.8 and Figure 6.3 show the results, while table 6.9 shows the mean and the standard deviation of the players' ratings after 5000 different ordering for the 86 played games.

	C_5-N0.001			C_5-N0.01			C_5-N0.0001		
	W	**D**	**L**	**W**	**D**	**L**	**W**	**D**	**L**
C_5-N0.001	-	-	-	41	15	30	45	12	29
C_5-N0.01				-	-	-	44	11	31

Table 6.8 Results when playing all C_5-N0.01, C_5-N0.001 and C_5-N0.0001 using the Two-Move Ballot.

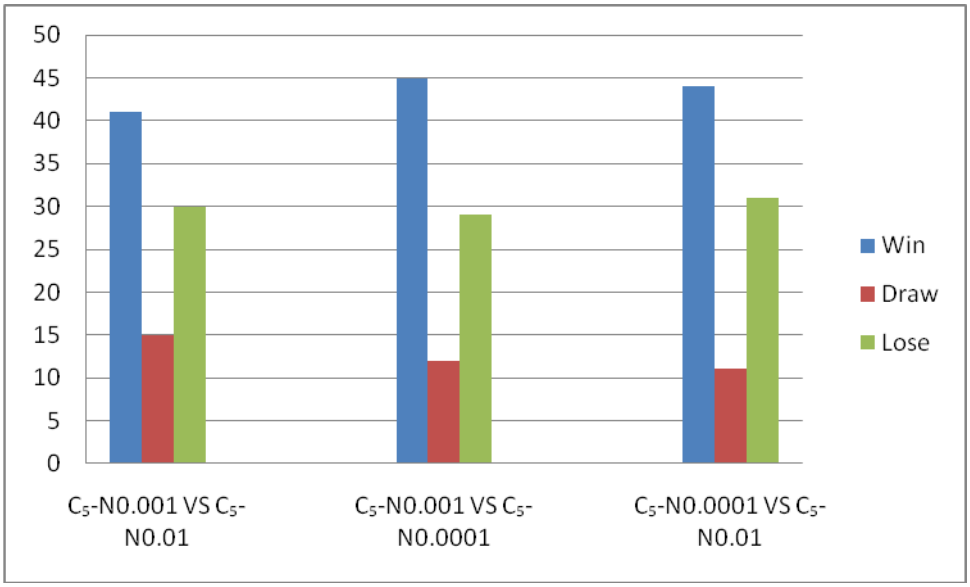

Figure 6.3 C_5-N0.01, C_5-N0.001 and C_5-N0.0001 against each other.

	Mean	SD	Class
C_5-N0.001	1228.28	30.75	D
C_5-N0.01	1182.09	29..48	E
C_5-N0.001	1438.72	30.90	C
C_5-N0.0001	1370.54	29.66	D
C_5-N0.01	1447.86	31.22	C
C_5-N0.0001	1393.70	29.94	D

Table 6.9 Standard rating formula for C_5-N0.01, C_5-N0.001 and C_5-N0.0001 against each other after 5000 ordering.

The results in tables 6.8 and 6.9 show that C_5-N0.001 is statistically better than C_5-N0.01 and C_5-N0.0001 as the results (when playing C_5-N0.001 against C_5-N0.01) put C_5-N0.001 in class D (rating = 1228) and put C_5-N0.01 in class E (rating = 1182), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_5-N0.001 and C_5-N0.01 are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha. The results (when playing C_5-N0.001 against C_5-N0.0001) put C_5-N0.001 in class C (rating = 1438) and put C_5-N0.0001 in class D (rating = 1370), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_5-N0.001 and C_5-N0.0001 are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha.

Finally the results in tables 6.8 and 6.9 show that C_5-N0.01 is statistically better than C_5-N0.0001 as the results (when playing C_5-N0.01 against C_5-N0.0001) put C_5-N0.01 in class C (rating = 1447) and put C_5-N0.0001 in class D (rating = 1393), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_5-N0.01 and C_5-N0.0001 are

statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha.

By using these results, it is clear that the C_5-N0.001 is better than both C_5-N0.01 and C_5-N0.0001 and thus the best value among the three selected ones is α =0.001.

To measure the effect of introducing 5-tuple as a learning method for the game of checkers, together with TDL, C_5-N0.001 was played against our four benchmark players (C_0, Blondie24-RR, C_{10}, and C_{10}-RR) using two-move ballot. The detailed results are in table 6.10 and figure 6.4.

	Opponent: C_5-N0.001		
	Win	Draw	Lose
C_0	30	10	46
Blondie24-RR	29	19	38
C_{10}	37	20	29
C_{10}-RR	49	16	21

Table 6.10 Results when Playing C_0, Blondie24-RR, C_{10} and C_{10}-RR against C_5-N0.001 using the Two-Move Ballot.

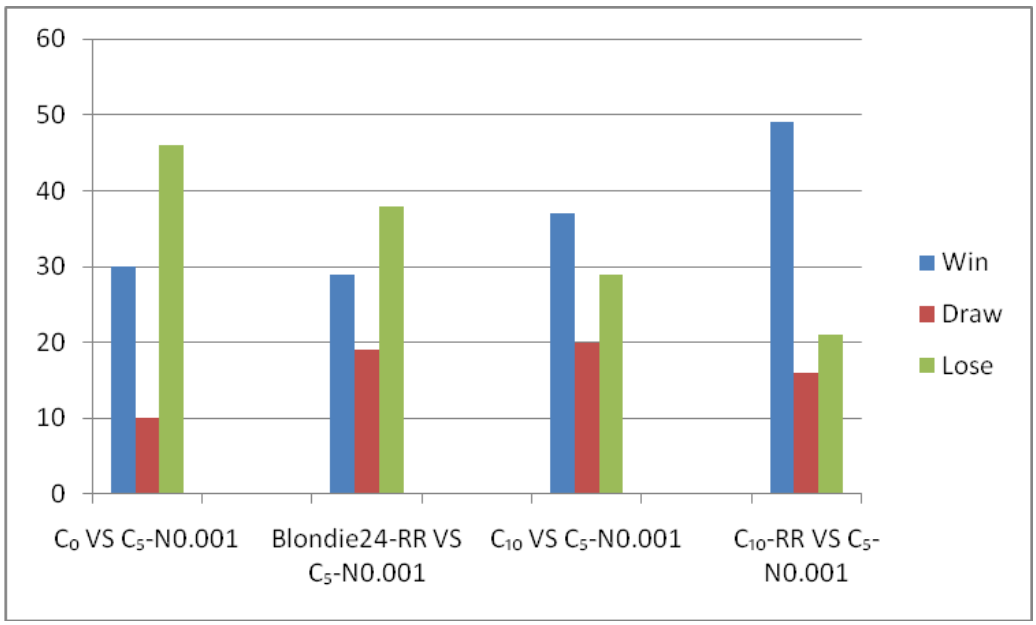

Figure 6.4 C_0, Blondie24-RR, C_{10} and C_{10}-RR against C_5-N0.001 using the Two-Move Ballot.

Table 6.11 summarises the results when playing against C_0, Blondie24-RR, C_{10} and against C_{10}-RR using a starting position where all pieces are in their original positions (i.e. no two-move ballot), while table 6.12 shows the mean and the standard deviation of the players' ratings after 5000 different ordering for the 86 played games.

		C_0	Blondie24-RR	C_{10}	C_{10}-RR
C_5-N0.001	Red	Won	Won	Lost	Lost
	White	Won	Won	Drawn	Lost

Table 6.11 Summary of Wins/Loses When not Using Two-Move Ballot.

	Mean	SD	Class
C_5-N0.001	1451.85	31.23	C
C_0	1382.87	29.97	D
C_5-N0.001	1209.79	29.54	D
Blondie24-RR	1169.43	28.32	E
C_5-N0.001	1255.27	28.01	D
C_{10}	1291.06	29.28	D
C_5-N0.001	1280.09	28.37	D
C_{10}-RR	1400.12	29.40	C

Table 6.12 Standard rating formula for C_5-tuple against C_0, Blondie24-RR, C_{10} and C_{10}-RR after 5000 ordering.

The results in tables 6.10 and 6.12 show that C_5-N0.001 is statistically better than C_0 as the results (when playing C_5-N0.001 against C_0) put C_5-N0.001 in class C (rating = 1451) and put C_0 in class D (rating = 1382), and by using student t-test (assuming unequal variances, $\alpha = 0.05$, and one-tail test), the results show that C_5-N0.001 and C_0 are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha.

The results in tables 6.10 and 6.12 also show that C_5-N0.001 is statistically better than Blondie24-RR as the results (when playing C_5-N0.001 against Blondie24-RR) put C_5-N0.001 in class D (rating = 1209) and put Blondie24-RR in class E (rating = 1169), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_5-N0.001 and Blondie24-RR are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha.

The results in tables 6.10 and 6.12 show that C_5-N0.001 and C_{10} are statistically the same as the results (when playing C_5-N0.001 against C_{10}) put C_5-N0.001 in class D (rating = 1255) and put C_{10} in class D (rating = 1291), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_5-N0.001 and C_{10} are statistically same as the P value (**P-value=0.5**) for the one tail t-test is greater than alpha.

Finally the results in tables 6.10 and 6.12 show that C_{10}-RR is statistically better than C_5-N0.001 as the results (when playing C_{10}-RR against C_5-N0.001) put C_{10}-RR in class C (rating = 1400) and put C_5-N0.001 in class D (rating = 1280), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_{10}-RR and C_5-N0.001 are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha.

The results in table 6.11 showed that C_5-N0.001 is better than C_0 and Blondie24-RR as C_5-N0.001 won as both red and white against them. Also the results in table 6.11 showed that C_5-N0.001 managed to get a draw as red against C_{10}.

Those results clearly validate our hypothesis (using an *n*-tuple architecture, together with TDL, will facilitate faster learning for the game of checkers and produce a good checkers player). As it took only two days to produce C_5-N0.001 (126,000 games), as opposed to the 19 days for the other versions.

The results in section table 6.11 showed that C_5-N0.001 lost as white against C_{10}. Also the results tables 6.11 and 6.12 clearly show that C_{10}-RR is better than C_5-N0.001. Although those results showed that C_5-N0.001 cannot beat C_{10} and C_{10}-RR but it was not our intention to produce a best player as this was not our hypothesis.

6.9 EXPERIMENTAL SETUP FOR 1-TUPLE WITH TDL

This section describes how to apply 1-tuple architecture to a checkers program with TDL. The difference with algorithm (6.3) are in steps 1, 3 and 5 (see algorithm 6.4), where a 5-tuple will be replaced by 1-tuple. Accordingly the number of weights will change.

1- **Take all the 32 possible checkers board squares. The n (n=1 for our experiments) positions can be arranged by choosing one square at a time.**

2- There is a one Look-Up Table (LUT) for each 5-Tuple.

3- **Since we have 5 types of pieces (our checker, our king, opponent's checker, opponent's king, and empty square), we require 5^1=5 possibilities for each n-tuple.**

4- The values for the pieces will be:-

- 0 for opponent's checker.
- 1 for opponent's king.
- 2 for Empty Square.
- 3 for our checker.

- 4 for our king.

5- **The total number of weights (32*5)=160, are initialized to zero.**

6- The result of evaluation the checkers board can be achieved by summing up all the corresponding LUT entries that are indexed by each n-tuple (in our case it will be only 32 entries each time).

7- In TDL the weights of the evaluation function are updated during game play using a gradient-descent method. Let x be the board observed by a player about to move, and similarly x′ the board after the player has moved. Then the evaluation function may be updated during play using equation (6.2).

8- If x' is a terminal state then the game has ended and using equation (6.3) for the update.

9- Repeat the process for G generations.

Algorithm 6.4 1-tuple for evolutionary checkers with TDL.

Three experiments were done, each one with different value for α (0.01, 0.001 and 0.001). Three players are constructed; those players are C_1-N0.01, C_1-N0.001 and C_1-N0.0001, where each player represents one of the selected values for α.

We run the algorithm for the same number of games (126,000, which requires about two days), required to produce C_0 (which took 19 days to evolve). All our experiments were run on the same computer (1.86 GHz Intel core2 processor and 2GB Ram).

6.10 RESULTS FOR 1-TUPLE WITH TDL

In order to determine which value is suitable for α from the three selected values, C_1-N0.01, C_1-N0.001 and C_1-N0.0001 were played against each other by using the idea of a two-move ballot. Table 6.13 and figure 6.5 show the results, while table 6.14 shows the mean and the standard deviation of the players' ratings after 5000 different ordering for the 86 played games.

	C_1-N0.001			C_1-N0.01			C_1-N0.0001		
	W	D	L	W	D	L	W	D	L
C_1-N0.001	-	-	-	43	14	29	40	20	26
C_1-N0.01				-	-	-	40	14	32

Table 6.13 Results when playing all C_1-N0.01, C_1-N0.001 and C_1-N0.0001 using the Two-Move Ballot.

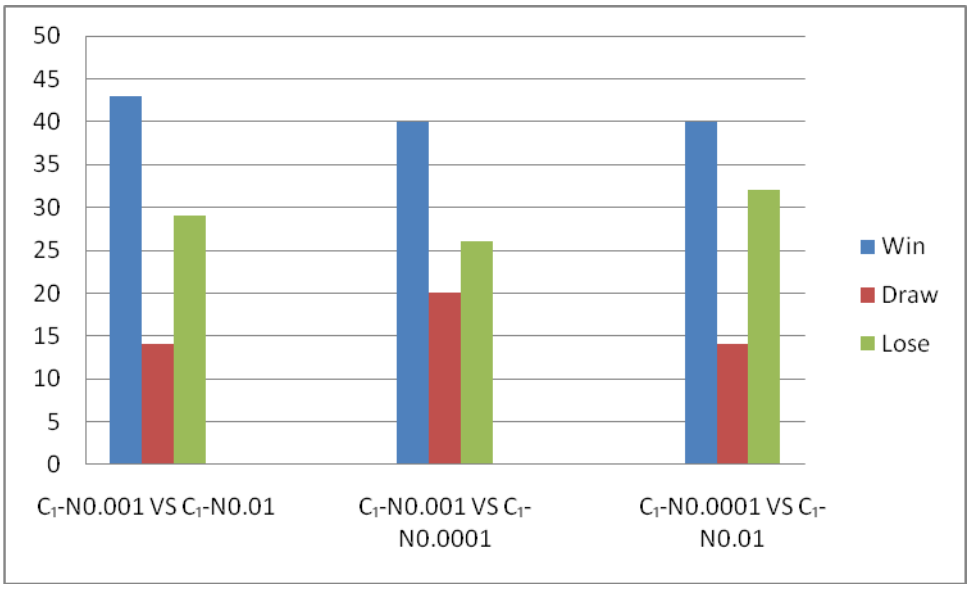

Figure 6.5 C_1-N0.01, C_1-N0.001 and C_1-N0.0001 against each other.

	Mean	SD	Class
C_1-N0.001	1401.31	30.64	C
C_1-N0.01	1342.21	29.40	D
C_1-N0.001	1254.37	29.12	D
C_1-N0.0001	1192.82	27.95	E
C_1-N0.01	1389.33	30.93	D
C_1-N0.0001	1353.43	29.63	D

Table 6.14 Standard rating formula for C_1-N0.01, C_1-N0.001 and C_1-N0.0001 against each other after 5000 ordering.

The results in tables 6.13 and 6.14 show that C_1-N0.001 is statistically better than C_1-N0.01 and C_1-N0.0001 as the results (when playing C_1-N0.001 against C_1-N0.01) put C_1-N0.001 in class C (rating = 1401) and put C_1-N0.01 in class D (rating = 1342), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_1-N0.001 and

C_1-N0.01 are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha. The results (when playing C_1-N0.001 against C_1-N0.0001) put C_1-N0.001 in class D (rating = 1254) and put C_1-N0.0001 in class E (rating = 1192), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_1-N0.001 and C_1-N0.0001 are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha.

Finally the results in tables 6.13 and 6.14 show that C_1-N0.01 and C_1-N0.0001 are statistically the same as the results (when playing C_1-N0.01 against C_1-N0.0001) put C_1-N0.01 in class D (rating = 1389) and put C_1-N0.0001 in class D (rating = 1353), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_1-N0.01 and C_1-N0.0001 are statistically same as the P value (**P-value=0.5**) for the one tail t-test is greater than alpha.

By using these results, it is clear that the C_1-N0.001 is better than both C_1-N0.01 and C_1-N0.0001 and thus the best value among the three selected ones is α =0.001.

To measure the effect of introducing 1-tuple as a learning method for the game of checkers, together with TDL, C_1-N0.001 was played against our four benchmark players (C_0, Blondie24-RR, C_{10}, and C_{10}-RR) using two-move ballot. The detailed results are in table 6.15 and figure 6.6.

	Opponent: C_1-N0.001		
	Win	Draw	Lose
C_0	28	19	39
Blondie24-RR	24	31	31
C_{10}	38	26	22
C_{10}-RR	54	12	20

Table 6.15 Results when Playing C_0, Blondie24-RR, C_{10} and C_{10}-RR against C_1-N0.001 using the Two-Move Ballot.

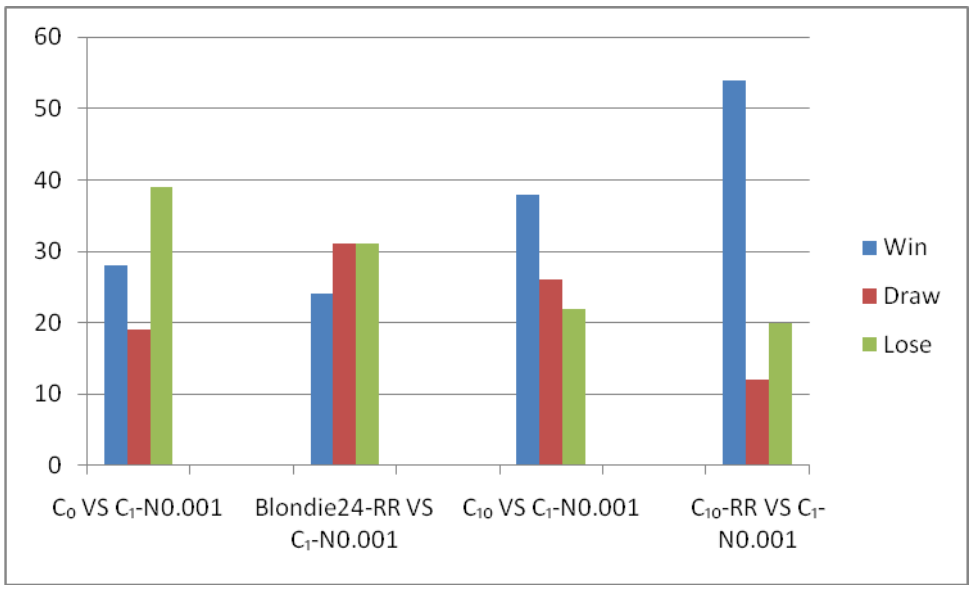

Figure 6.6 C_0, Blondie24-RR, C_{10} and C_{10}-RR against C_5-N0.001 using the Two-Move Ballot.

Table 6.16 summarises the results when playing against C_0, Blondie24-RR, C_{10} and against C_{10}-RR using a starting position where all pieces are in their original positions (i.e. no two-move ballot), while table 6.17 shows the mean and the standard deviation of the players' ratings after 5000 different ordering for the 86 played games.

		C_0	Blondie24-RR	C_{10}	C_{10}-RR
C_1-N0.001	**Red**	Won	Won	Lost	Lost
	White	Won	Won	Lost	Lost

Table 6.16 Summary of Wins/Loses When not Using Two-Move Ballot.

	Mean	SD	Class
C_1-N0.001	1314.14	29.62	D
C_0	1265.35	28.41	D
C_1-N0.001	1108.78	27.25	E
Blondie24-RR	1077.25	26.11	E
C_1-N0.001	1140.79	26.97	E
C_{10}	1210.30	28.04	D
C_1-N0.001	1331.53	29.06	D
C_{10}-RR	1480.38	30.07	C

Table 6.17 Standard rating formula for C_1-tuple against C_0, Blondie24-RR, C_{10} and C_{10}-RR after 5000 ordering.

The results in tables 6.15 and 6.17 show that C_1-N0.001 and C_0 are statistically the same as the results (when playing C_1-N0.001 against C_0) put C_1-N0.001 in class D (rating = 1314) and put C_0 in class D (rating = 1265), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_1-N0.001 and C_0 are statistically same as the P value (**P-value=0.5**) for the one tail t-test is greater than alpha.

The results in tables 6.15 and 6.17 show that C_1-N0.001 and Blondie24-RR are statistically the same as the results (when playing C_1-N0.001 against Blondie24-RR) put C_1-N0.001 in class E (rating = 1108) and put Blondie24-RR in class E (rating = 1077), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_1-N0.001 and Blondie24-RR are statistically same as the P value (**P-value=0.5**) for the one tail t-test is greater than alpha.

The results in tables 6.15 and 6.17 also show that C_{10} is statistically better than C_1-N0.001 as the results (when playing C_{10} against C_1-N0.001) put C_{10} in class D (rating = 1210) and put C_1-N0.001 in class E (rating = 1140), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail

test), the results show that C_{10} and C_1-N0.001 are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha.

Finally the results in tables 6.15 and 6.17 show that C_{10}-RR is statistically better than C_1-N0.001 as the results (when playing C_{10}-RR against C_1-N0.001) put C_{10}-RR in class C (rating = 1480) and put C_1-N0.001 in class D (rating = 1331), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_{10}-RR and C_1-N0.001 are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha.

The results in section 6.10 showed that C_1-N0.001 is statistically the same as C_0 and Blondie24-RR. Those results clearly validate our hypothesis (using an *n*-tuple architecture, together with TDL, will facilitate faster learning for the game of checkers and produce a good checkers player). As it took only two days to produce C_1-N0.001 (126,000 games), whereas it took 19 days for our four other baseline players.

The results in section 6.10 clearly show that C_{10} and C_{10}-RR are better than C_1-N0.001. Although those results showed that C_1-N0.001 cannot beat C_{10} and C_{10}-RR but it was not our intention to produce a best player as this was not our hypothesis.

The results in sections 6.8, and 6.10 show that C_5-N0.001 is almost the same as C_1-N0.001. To be more confident about that, we decided to play C_5-N0.001 and C_1-N0.001 using the two move ballot. The next section shows the results, which is obtained using a single TDL run.

6.11 C_5-N0.001 Against C_1-N0.001

	Opponent: C_1-N0.001		
	Win	Draw	Lose
C_5-N0.001	29	35	22

Table 6.18 Results when Playing C_5-N0.001 against C_1-N0.001 using the Two-Move Ballot.

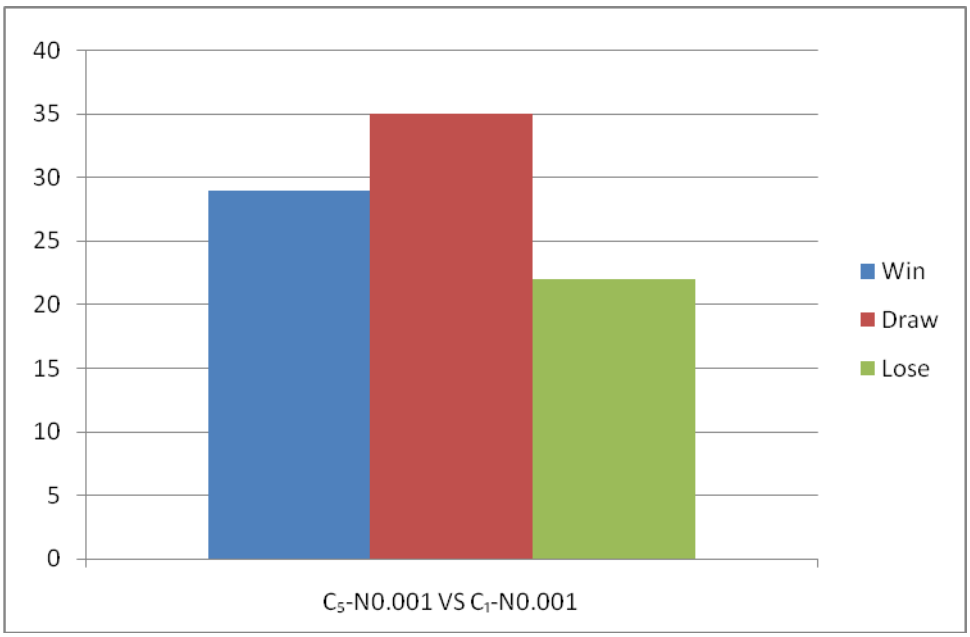

Figure 6.7 C_5-N0.001 against C_1-N0.001.

The results of the 5000 different orderings show that C_5-N0.001 and C_1-N0.001 are statistically the same, as the results put C_5-N0.001 in class E (rating = 1043) and put C1-N0.001 in class E (rating =1011), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_5-N0.001 and C_1-N0.001 are statistically same as the P value (**P-value=0.5**) for the one tail t-test is greater than alpha.

6.12 SUMMARY

This chapter has introduced *n*-tuples to the game of checkers. Two main experiments were carried out. The first used the *n*-tuple with an evolutionary checkers based on the architecture of Blondie24. Sections 6.3 and 6.4 showed the experimental setup and the results of using 5-tuples with a random walk. Sections 6.5 and 6.6 showed the experimental setup and the results for using a 1-tuple.

The results demonstrated that using 5-tuples with a random walk and a 1-tuple did not evolve a good player as both the results for 5-tuples with random walk in section 6.4 and the results for a 1-tuple in section 6.6 showed that C_5-tuple and C_1-tuple are not good players when compared with C_0, Blondie24-RR, C_{10} and C_{10}-RR.

The second experiment used both 5-tuples with random walking and a 1-tuple with TDL. The proposed algorithm in sections 6.7 and 6.9 showed promising results when tested against C_0, Blondie-RR, C_{10} and C_{10}-RR. The players were trained in a time that is much less than the time required evolving C_0, Blondie24-RR, C_{10} and C_{10}-RR (2 days compared with 19 days). The results in section 6.11 showed that the 5-tuples with random walks player is statistically same 1-tuple player.

In summary, the combination of temporal difference learning with *n*-tuples seems a very promising approach.

The experiments also showed that the best value for α from the three selected values is 0.001.

Based on the results we obtained it would seem appropriate to use *n*-tuple learning to enhance the ability of the constructed self learning computer checkers players.

The next chapter will discuss the importance of piece difference feature together with the importance of a look-ahead feature for evolutionary checkers.

Chapter Seven

The Importance of Piece Difference Feature and Look-ahead Depth to Evolutionary Checkers

7.1 INTRODUCTION

Chapter six showed that using *n*-tuple systems within an evolutionary checkers framework produced a good player, considerably quicker time than previous approaches. This chapter investigates the importance of the piece difference feature and the look-ahead depths for evolutionary computer checkers. Therefore we will investigate evolutionary neural networks, with and without piece difference, and with different ply depths, evolved via an evolution strategy. We believe that those two features are important in the design of evolutionary computer checkers but we would like to investigate this aspect of the framework.

This chapter is structured as follows; Section 6.3 describes the experiments that were done by Fogel and Hughes to show the importance of piece difference for the design of Blondie24. Sections 7.3 and 7.4 describe the experimental setup and the results for the piece difference. In section 7.5 we start to investigate the effect of the look-ahead depth. Sections 7.6 and

7.7 describe the experimental setup and the results for the look-ahead depth. Finally a summary for this chapter is presented in section 7.8. This chapter has been disseminated via the following publications: Al-Khateeb and Kendall (2010, 2011c).

7.2 PIECE DIFFERENCE

Blondie24 represents a landmark in evolutionary learning. Even so, it has still attracted comments about its design. One of them is concerned with the piece difference (The difference of the number of the player pieces currently on the board and the number of the opponent pieces currently on the board) feature and how it affects the learning process of Blondie24. This was answered (by Fogel) by playing a series of fourteen matches (seven as red and seven as white) between Blondie24 and a piece-count player (Chellapilla and Fogel 1999 and Fogel 2002). The experiment showed that the piece-count player played a weak endgame, because it is unable to see far enough ahead to capture a piece. The games played out until either the game was completed (with one side winning, or a draw being declared due to the number of repeated positions). In the case of a draw an assessment of the outcome was made by examining the piece advantage that one player had over the other, and also by playing out the game using a strong computer program (Blitz98), which played out the remainder of the game and declared a winner.

Of the fourteen games played, two were played to completion, with Blondie24 winning both. For the remaining twelve games, Blondie24 held an

advantage in ten games, while the piece-count player held the advantage in two games (see Table 7.1). By using Blitz98 to play out the twelve incomplete games, Blondie24 got wins in eight games; the piece-count player won one game, while the remaining three games ended in a draw (Table 7.2).

	Piece-count		
	Win	Draw	Lose
Blondie24	12	0	2

Table 7.1 Results of Playing 14 Games between Blondie24 and Piece-count Using Material Advantage to Break Tie.

	Piece-count		
	Win	Draw	Lose
Blondie24	10	3	1

Table 7.2 Results of Playing 14 Games between Blondie24 and Piece-count Using Blitz98 to Break Tie.

It is clear from Table 7.1 and 7.2 that Blondie24 is better than a piece-count player, and by using a standard rating formula, the results suggest that Blondie24 is about 311 to 400 points better than the piece-count player based on material advantage or the final outcome using Blitz98 (Chellapilla and Fogel 1999 and Fogel 2002).

The results demonstrate that a piece difference feature is important to Blondie24 but the neural network has additional information that is important to learning within Blondie24 (Fogel 2002).

Hughes also wanted to investigate the importance of a piece difference feature to the design of Brunette24 (a re-implementation of Blondie24) by evolving a piece difference heuristic using co-evolution (Hughes 2003).

Hughes used the same experiment as Fogel to show the importance of a piece difference. This was done by playing 1000 games against a simple piece difference player. The evolved piece difference player managed to win 68% of the games, drew 30% and lost 2% (Table 7.3).

Also, to measure the success of the evolved piece difference player, Hughes played 1000 games against xcheckers, which is free software available from http://arton.cunst.net/xcheckers. The evolved piece difference player won 22% of the games, drew 66% and lost 12% (Table 7.4).

It is worth to mention that the evolved piece count player is constructed by giving each board location a weight. The evolved piece count player is also called a weighted piece count player.

	Piece-count		
	Win	Draw	Lose
Evolved Piece Count	680	300	20

Table 7.3 Results of Playing 1000 Games between the Evolved Piece Count player and Piece-count player.

	Xcheckers		
	Win	Draw	Lose
Evolved Piece Count	220	660	120

Table 7.4 Results of Playing 1000 Games between the Evolved Piece Count player and xcheckers.

The results in Table 7.3 show that the evolved piece count player is better than the piece count player and by using a standard rating formula, the results suggest that the evolved piece difference player is about 528 points better than piece difference player. While applying the standard rating formula to the results in Table 7.4 shows that the evolved piece difference player is about 80 points better than xcheckers. These results, like Fogel's, also show that a piece difference feature is important.

7.3 EXPERIMENTAL SETUP FOR PIECE DIFFERENCE

Our hypothesis is that the piece difference feature is important to evolutionary checkers and this can be done by evolving two players, one with the piece difference feature and the other without the piece difference feature. The work carried out here is differs to the work of Fogel and Hughes in that they used a piece count player and evolve a player with piece difference feature. Two evolutionary checkers players were implemented; one with a piece difference feature, which is called C_0, while the other is without a piece difference feature and is called C_0-NPD. The implementation for both players is based on algorithm 3.1.

Our previous efforts to enhance evolutionary checkers introduced a round robin tournament (see chapter four). We decided to use the resultant player (Blondie24-RR) to show the importance of the piece difference feature. This is done by implementing a player which is the same as Blondie24-RR, but, does not include a piece difference feature. This player is called Blondie24-RRNPD.

Also, we decided to show the importance of the piece difference feature in evolutionary checkers with individual and social learning and with *n*-tuple systems.

In this case C_{10}, the resultant player from introducing individual and social learning (chapter five), will be used to show the importance of the piece difference feature. This is done by implementing a player which is the same as C_{10}, but, does not include a piece difference feature. This player is called C_{10}-NPD.

Also C_5-N0.001, the resultant player from introducing *n*-tuple systems (chapter six), will be used to show the importance of the piece difference feature. This is done by implementing a player which is the same as C_5-N0.001, but, does not include a piece difference feature. This player is called C_5-N0.001-NPD.

7.4 RESULTS FOR PIECE DIFFERENCE

To measure the effect of a piece difference feature in evolutionary checkers, C_0 was played against C_0-NPD by using the idea of a two-move ballot. We play all of the 43 possible games, both as red and white. This gives a total of 86 games. The games were played until either one side wins or a draw is declared after 100 moves for each player. The same procedure was also used to play Blondie24-RR against Blondie24-RRNPD, C_{10} against C_{10}-NPD and C_5-N0.001against C_5-N0.001-NPD. The results are shown in Tables 7.5 through 7.8 and in Figure 7.1.

	Opponent:C_0-NPD		
	Win	Draw	Lose
C_0	59	14	13

Table 7.5 Results when Playing C_0 against C_0-NPD using the Two-Move Ballot.

	Opponent: Blondie24-RRNPD		
	Win	Draw	Lose
Blondie24-RR	61	16	9

Table 7.6 Results when Playing Blondie24-RR against Blondie24-RRNPD using the Two-Move Ballot.

	Opponent:C_{10}-NPD		
	Win	Draw	Lose
C_{10}	55	16	15

Table 7.7 Results when Playing C_{10} against C_{10}-NPD using the Two-Move Ballot.

	Opponent:C_5-N0.001-NPD		
	Win	Draw	Lose
C_5-N0.001	60	12	14

Table 7.8 Results when Playing C_5-N0.001 against C_5-N0.001-NPD using the Two-Move Ballot.

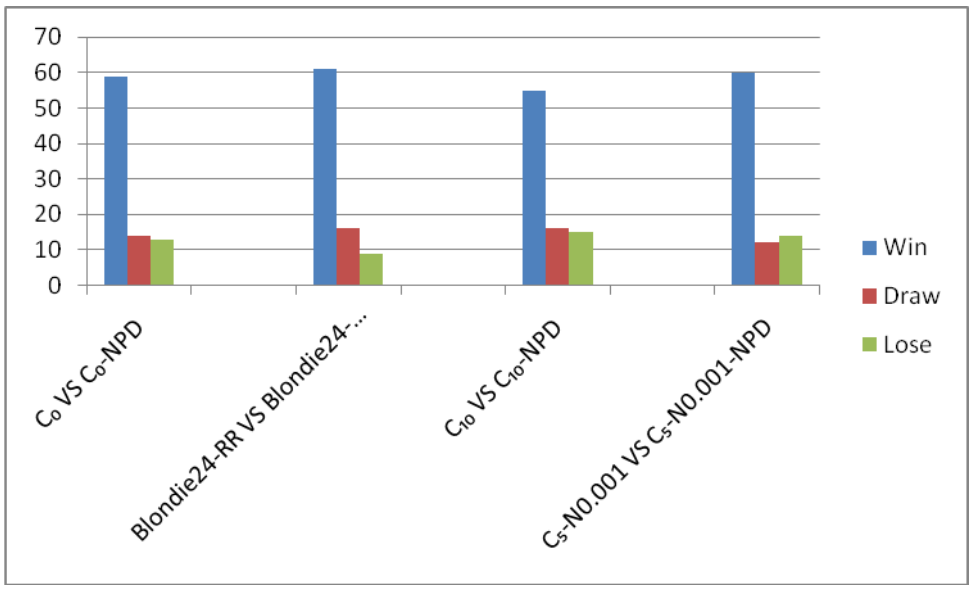

Figure 7.1 C_0 against C_0-NPD, Blondie24-RR against Blondie24-RRNPD, C_{10} against C_{10}-NPD and C_5-N0.001against C_5-N0.001-NPD.

Table 6.9 summarises the results when the players playing using a starting position where all pieces are in their original positions (i.e. no two-move ballot), while table 6.10 shows the mean and the standard deviation of the players' ratings after 5000 different ordering for the 86 played games.

		C_0-NPD	Blondie24-RRNPD	C_{10}-NPD	C_5-N0.001-NPD
C_0	**Red**	Won	-	-	-
	White	Won	-	-	-
Blondie24-RR	**Red**	-	Won	-	-
	White	-	Won	-	-
C_{10}	**Red**	-	-	Won	-
	White	-	-	Won	-
C_5-N0.001	**Red**	-	-	-	Won
	White	-	-	-	Won

Table 7.9 Summary of Wins/Loses When not Using Two-Move Ballot.

	Mean	SD	Class
C_0	1481.25	27.40	C
C_0-NPD	1267.42	26.73	D
Blondie24-RR	1466.79	25.05	C
Blondie24-RRNPD	1217.84	24.60	D
C_{10}	1431.34	27.27	C
C_{10}-NPD	1251.00	26.51	D
C_5-N0.001	1512.50	28.08	C
C_5-N0.001NPD	1301.35	27.36	D

Table 7.10 Standard rating formula for C_0 against C_0-NPD, Blondie24-RR against Blondie24-RRNPD, C_{10} against C_{10}-NPD and C_5-N0.001against C_5-N0.001-NPD.after 5000 ordering.

The results in tables 7.5 and 7.10 show that C_0 is statistically better than C_0-NPD as the results (when playing C_0 against C_0-NPD) put C_0 in class C (rating = 1481) and put C_0-NPD in class D (rating = 1267), and by using student t-test (assuming unequal variances, $\alpha = 0.05$, and one-tail test), the

results show that C_0 and C_0-NPD are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha.

The results in tables 7.6 and 7.10 also show that Blondie24-RR is statistically better than Blondie24-RRNPD as the results (when playing Blondie24-RR against Blondie24-RRNPD) put Blondie24-RR in class C (rating = 1466) and put Blondie24-RRNPD in class D (rating = 1217), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that Blondie24-RR and Blondie24-RRNPD are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha.

The results in tables 7.7 and 7.10 also show that C_{10} is statistically better than C_{10}-NPD as the results (when playing C_{10} against C_{10}-NPD) put C_{10} in class C (rating = 1431) and put C_{10}-NPD in class D (rating = 1251), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_{10} and C_{10}-NPD are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha.

Finally the results in tables 7.8 and 7.10 show that C_5-N0.001 is statistically better than C_5-N0.001-NPD as the results (when playing C_5-N0.001 against C_5-N0.001-NPD) put C_5-N0.001 in class C (rating = 1512) and put C_5-N0.001-NPD in class D (rating = 1301), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_5-N0.001 and C_5-N0.001-NPD are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha.

Using the results from our experiments and those of Fogel's and Hughes', we can conclude that a piece difference feature is important to the design of the evolutionary checkers. Of course, the neural network is also an important element of the whole design but the results presented here demonstrate a simple feature is able to significantly improve the overall playing strength.

Now that the importance of piece difference has been shown in the design of the evolutionary checkers, the next sections will investigate if the depth of the search is also an important element. We suspect that it is, but we would like to investigate this aspect of the framework.

7.5 LOOK-AHEAD

There has been a lot of discussion about the importance of the look-ahead depth level used in Fogel's work: There is little work that has rigorously investigated its importance. Fogel, in his work on evolving Blondie24 (Fogel 2002), showed the importance of using a four ply search in Blondie24 by stating that "*At four ply, there really isn't any "deep" search beyond what a novice could do with a paper and pencil if he or she wanted to*". In fact we don't believe that this is the case as generating all the possible moves from a four ply search is not an easy task for novices, and would also be time consuming. Of course, it might be done at some subconscious level, where pruning is taking place, but this (as far as we are aware) has not been reported in the scientific literature.

Many researchers have shown the importance of the look-ahead depth for computer games, but none of them was related to checkers. Most of the findings are related to chess (Bettadapur and Marsland 1988, Levene and Fenner 2001, Nau et.al. 2001 and Smet et. al. 2003), where it was shown that increasing the depth level will produce superior chess players. However, (Runarsson and Jonsson 2007) showed that this was not the case for Othello, as they found that better playing strategies are found when TD learning with ε–greedy is applied with a lower look-ahead search depth and a deeper look-ahead search during game play. Given that chess appears to benefit from a deeper look-ahead, but this is not true for Othello, this chapter will establish if checkers benefits from a deeper look-ahead.

7.6 EXPERIMENTAL SETUP FOR LOOK-AHEAD DEPTH

Our hypothesis is that the look-ahead depth feature is important to evolutionary checkers and this can be done by evolving evolutionary checkers player, based on the same algorithm that was used to construct Blondie24. Our implementation has the same structure and architecture that Fogel utilised in Blondie24.

Four implementations were made; those players are listed below:-

1- C1Ply trained using one ply depth.

2- C2Ply trained using two ply depth.

3- C3Ply trained using three ply depth.

4- C4Ply trained using four ply depth.

Each player played against all other players but was now allowed to search to a depth of 6-ply. The reason to choose 6-ply is to allow each program to search in a level that is greater than the level trained with, which is the case that Fogel used in Blondie24 (trained at 4-ply and played at a higher level).

Our previous efforts to enhance evolutionary checkers introduced a round robin tournament (Chapter Four). We also use this player (Blondie24-RR) to investigate the importance of the look-ahead depth. This is done by implementing three other players, which are the same as Blondie24-RR, but, trained on different ply depths, those players are called:-

1- Blondie24-RR1Ply.

2- Blondie24-RR2Ply.

3- Blondie24-RR3Ply

It is worth mentioning that Blondie24-RR is constructed using a four ply depth. Each player was set to play against all the other three players but now using a six ply depth.

7.7 RESULTS FOR LOOK-AHEAD DEPTH

In order to provide a fair comparison, all the experiments were run using same computer (1.86 GHz Intel core2 processor and 2GB Ram). All the experiments to evolve the players were run for the same number of generations (840 and 126,000 played games). The following subsections show the results for all the constructed players.

7.7.1 Results for C1Ply, C2Ply, C3Ply and C4Ply

To measure the effect of increasing the ply depth in the game of checkers, each player trained at a given ply was matched with all of the other players trained with on different ply. A league is held between C1Ply, C2Ply, C3Ply and C4Ply; each match in the league was played using the idea of a two-move ballot. For each match we play all of the 43 possible games, both as red and white. This gives a total of 86 games. The total number of games played is 258. Each game is played using a fixed ply depth of six. The games were played until either one side wins or a draw is declared after 100 moves for each player. The results are shown in tables 7.11 through 7.13 and in figure 7.2.

	C1Ply	C2Ply	C3Ply	C4Ply	Σwins
C1Ply	-	28	17	13	58
C2Ply	33	-	24	19	76
C3Ply	45	31	-	27	103
C4Ply	59	40	35	-	134

Table 7.11 Number of wins (for the row player) out of 258 games.

	C1Ply	C2Ply	C3Ply	C4Ply	Σdraws
C1Ply	-	25	24	14	63
C2Ply	25	-	31	27	83
C3Ply	24	31	-	26	91
C4Ply	14	27	26	-	67

Table 7.12 Number of draws (for the row player) out of 258 games.

	C1Ply	C2Ply	C3Ply	C4Ply	Σlosses
C1Ply	-	33	45	59	137
C2Ply	28	-	31	40	99
C3Ply	17	24	-	33	74
C4Ply	13	19	27	-	59

Table 7.13 Number of losses (for the row player) out of 258 games.

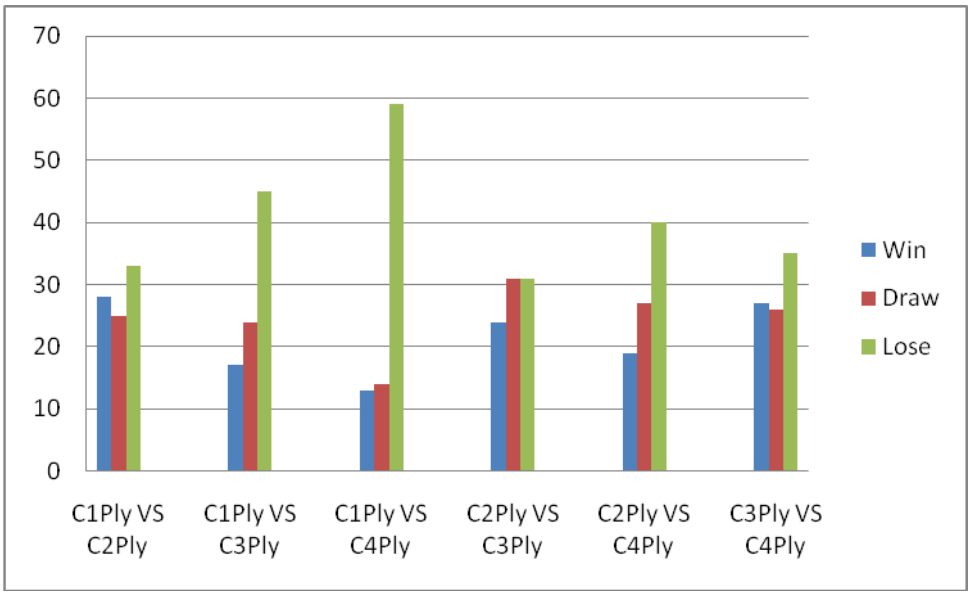

Figure 7.2 Results of playing a league between C1Ply, C2Ply, C3Ply and C4Ply.

It is clear from tables 7.11 and 7.13 that the total number of wins increases and the total number of losses decreases when the evolved ply depth increases. Therefore, increasing the ply depth leads to a superior player. Table 7.14 shows the mean and the standard deviation of the players' ratings after 5000 different orderings for the 86 played games, while table 7.15 summarises the results when playing the league between players using a starting position where all pieces are in their original positions (i.e. no two-move ballot).

	Mean	SD	Class
C1Ply	1188.94	28.94	E
C2Ply	1206.24	27.62	D
C1Ply	1146.58	27.40	E
C3Ply	1266.18	26.14	D
C1Ply	1264.11	27.21	D
C4Ply	1474.99	26.14	C
C2Ply	1179.47	26.85	E
C3Ply	1205.10	25.60	D
C2Ply	1114.61	27.17	E
C4Ply	1200.21	25.88	D
C3Ply	1176.02	28.26	E
C4Ply	1205.26	26.98	D

Table 7.14 Standard rating formula for all players after 5000 different orderings of the 86 games played.

		C1Ply	C2Ply	C3Ply	C4Ply
C1Ply	**Red**	-	Lost	Lost	Lost
	White	-	Drawn	Lost	Lost
C2Ply	**Red**		-	Lost	Lost
	White		-	Drawn	Lost
C3Ply	**Red**			-	Lost
	White			-	Lost

Table 7.15 Summary of Wins/Loses for C1Ply, C2Ply, C3Ply and C4Ply When not Using Two-Move Ballot.

The results in table 7.14, obtained using 5000 different orderings for the 86 games (obtained using the two-move ballot) show that increasing ply depth by one increases the performance of the checkers player as C2Ply is better (using our definition given earlier with respect to players having a different rating class) than C1Ply, C3Ply is better than C2Ply and C4 is better than C3Ply, and by using the average value for the standard rating formula the results (when playing C2Ply against C1Ply) put C2Ply in class D (rating = 1206) and put C1Ply in Class E (rating = 1189), and by using student t-test (assuming unequal variances, $\alpha = 0.05$, and one-tail test), the results show

that C1Ply and C2Ply are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha.

Also the results (when playing C3Ply against C2Ply) in table 7.14 put C3Ply in class D (rating = 1205) and put C2Ply in class E (rating = 1179), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C2Ply and C3Ply are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha and finally (when playing C4Ply against C3Ply) put C4Ply in class D (rating = 1205) and put C3Ply in class E (rating = 1176), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C3Ply and C4Ply are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha.

The results shown in table 7.14 also show that increasing ply depth by two increases the performance of the checkers player as C3Ply and C4Ply are significantly better than the C1Ply and C2Ply respectively, and by using the average value for the standard rating formula, the results (when playing C3Ply against C1Ply) put C3Ply in class D (rating = 1266) and C1Ply in Class E (rating = 1147), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C1Ply and C3Ply are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha, while (when playing C4Ply against C2Ply), C4Ply is in Class D (rating = 1200) and C2Ply is in class E (rating = 1115), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results

show that C2Ply and C4Ply are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha.

Finally the results in table 7.14 show that C4Ply is significantly better than the C1Ply, and by using the average value for the standard rating formula, the results (when playing C4Ply against C1Ply) puts C4Ply in class C (rating = 1475) and C1Ply in class D (rating = 1264), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C1Ply and C4Ply are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha.

7.7.2 Results Using Round Robin Players

The same procedure in section 7.6.1 was also used to play a league between Blondie24-RR, Blondie24-RR1Ply, Blondie24-RR2Ply and Blondie24-RR3Ply. The results are shown in tables 7.16 through 7.18 and figure 7.3.

	Blondie24-RR1Ply	**Blondie24-RR2Ply**	**Blondie24-RR3Ply**	**Blondie24-RR**	**Σ wins**
Blondie24-RR1Ply	-	28	20	14	62
Blondie24-RR2Ply	32	-	29	21	82
Blondie24-RR3Ply	42	34	-	27	103
Blondie24-RR	57	46	39	-	142

Table 7.16 Number of wins (for the row player) out of 258 games for the round robin players.

	Blondie24-RR1Ply	Blondie24-RR2Ply	Blondie24-RR3Ply	Blondie24-RR	Σ draws
Blondie24-RR1Ply	-	26	24	15	65
Blondie24-RR2Ply	26	-	23	19	68
Blondie24-RR3Ply	24	23	-	20	67
Blondie24-RR	15	19	20	-	54

Table 7.17 Number of draws (for the row player) out of 258 games for the round robin players.

	Blondie24-RR1Ply	Blondie24-RR2Ply	Blondie24-RR3Ply	Blondie24-RR	Σ losses
Blondie24-RR1Ply	-	32	42	57	131
Blondie24-RR2Ply	28	-	34	46	108
Blondie24-RR3Ply	20	29	-	39	88
Blondie24-RR	14	21	27	-	62

Table 7.18 Number of losses (for the row player) out of 258 games for the round robin players.

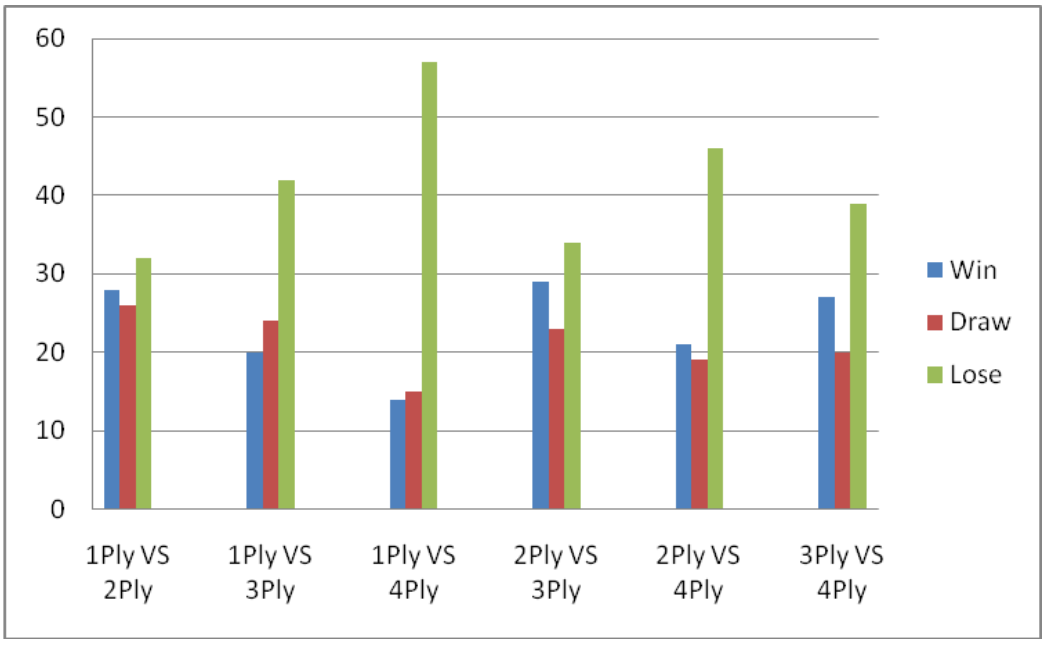

Figure 7.3 Results of playing a league between Blondie24-RR1Ply, Blondie24-RR2Ply, Blondie24-RR3Ply and Blondie24-RR.

It is clear from tables 7.16 and 7.18 that the total number of wins increases and the total number of losses decreases when the ply depth increases. Therefore, increasing the ply depth leads to a superior player. Table 7.19 shows the mean and the standard deviation of the players' ratings after 5000 different orderings for the 86 played games, while table 7.20 summarises the results when playing the league between players using a starting position where all pieces are in their original positions (i.e. no two-move ballot).

	Mean	SD	Class
Blondie24-RR1Ply	1187.79	28.86	E
Blondie24-RR 2Ply	1200.74	27.55	D
Blondie24-RR 1Ply	1160.17	28.15	E
Blondie24-RR 3Ply	1252.67	26.84	D
Blondie24-RR 1Ply	1256.00	27.71	D
Blondie24-RR	1450.51	26.58	C
Blondie24-RR 2Ply	1194.62	29.30	E
Blondie24-RR 3Ply	1212.04	27.98	D
Blondie24-RR 2Ply	1335.38	28.72	D
Blondie24-RR	1440.84	27.43	C
Blondie24-RR 3Ply	1348.31	29.24	D
Blondie24-RR	1495.93	27.91	C

Table 7.19 Standard rating formula for all players after 5000 different orderings of the 86 games played.

		Blondie24-RR1Ply	Blondie24-RR2Ply	Blondie24-RR3Ply	Blondie24-RR
Blondie24-RR1Ply	**Red**	-	Lost	Lost	Lost
	White	-	Lost	Lost	Lost
Blondie24-RR2Ply	**Red**		-	Lost	Lost
	White		-	Lost	Lost
Blondie24-RR3Ply	**Red**			-	Lost
	White			-	Lost

Table 7.20 Summary of Wins/Loses for Blondie24-RR1Ply, Blondie24-RR2Ply, Blondie24-RR3Ply and Blondie24-RR When not Using Two-Move Ballot.

The results in table 7.19, obtained using 5000 different orderings for the 86 games (obtained using the two-move ballot) show that increasing depth by one increases the performance of the checkers player as Blondie24-RR2Ply is better than the Blondie24-RR1Ply, Blondie24-RR3Ply is better than Blondie24-RR2Ply and Blondie24-RR is better than Blondie24-RR3Ply. By using the average value for the standard rating formula the results (when playing Blondie24-RR2Ply against Blondie24-RR1Ply) put Blondie24-RR2Ply in class D (rating = 1201) and Blondie24-RR1Ply in Class E (rating = 1188), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that Blondie24-RR1Ply and Blondie24-RR2Ply are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha. Playing Blondie24-RR3Ply against Blondie24-RR2Ply puts Blondie24-RR3Ply in class D (rating = 1212) and Blondie24-RR2Ply in class E (rating = 1195), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that Blondie24-RR2Ply and Blondie24-RR3Ply are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha. Finally, when playing Blondie24-RR against Blondie24-RR3Ply, puts Blondie24-RR in class C (rating = 1496) and Blondie24-RR3Ply in class D (rating = 1348), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that Blondie24-RR3Ply and Blondie24-RR are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha..

The results shown in table 7.19 also show that increasing depth by two increases the performance of the checkers player as Blondie24-RR3Ply and

Blondie24-RR are significantly better than the Blondie24-RR1Ply and Blondie24-RR2Ply respectively, and by using the average value for the standard rating formula, the results (when playing Blondie24-RR3Ply against Blondie24-RR1Ply) put Blondie24-RR3Ply in class D (rating = 1253) and Blondie24-RR1Ply in class E (rating = 1160), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that Blondie24-RR1Ply and Blondie24-RR3Ply are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha, while (when playing Blondie24-RR against Blondie24-RR2Ply) puts Blondie24-RR in Class C (rating = 1441) and Blondie24-RR2Ply in class D (rating = 1335), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that Blondie24-RR2Ply and Blondie24-RR are statistically different as the P value (**P-value=0**) for the one tail t-test is less than alpha.

Finally the results in table 7.19 show that Blondie24-RR is significantly better than the Blondie24-RR1Ply, and by using the average value for the standard rating formula, the results (when playing Blondie24-RR against Blondie24-RR1Ply) puts Blondie24-RR in class C (rating = 1450) and Blondie24-RR1Ply in class D (rating = 1256), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that Blondie24-RR1Ply and Blondie24-RR are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha.

Using the results from our experiments, we can conclude that a look-ahead depth is important to the design of the evolutionary checkers. Also

additional experiments are done to check if this is the case for evolutionary checkers with individual and social learning and *n*-tuple systems. Section 7.7.3 shows the results.

7.7.3 Results Using Individual and Social Learning Players and *N*-tuple Players

Two individual and social learning players were constructed; C_{10}-4Ply, which is trained with four ply depth and C_{10}-1Ply, which is trained using only one ply depth. Those players will play against each other using two-move ballot. Two *n*-tuples players were also constructed; C_5-N0.001-4Ply, which is trained with four ply depth and C_5-N0.001-1Ply, which is trained using only one ply depth. Those players will play against each other using two-move ballot. Tables 7.21 and 7.22 and figure 7.4 show the results.

	Opponent:C_{10}-1Ply		
	Win	Draw	Lose
C_{10}-4Ply	56	16	14

Table 7.21 Results when Playing C_{10}-4Ply against C_{10}-1Ply using the Two-Move Ballot.

	Opponent:C_5-N0.001-1Ply		
	Win	Draw	Lose
C_5-N0.001-4Ply	51	20	15

Table 7.22 Results when Playing C_5-N0.001-4Ply against C_5-N0.001-1Ply using the Two-Move Ballot.

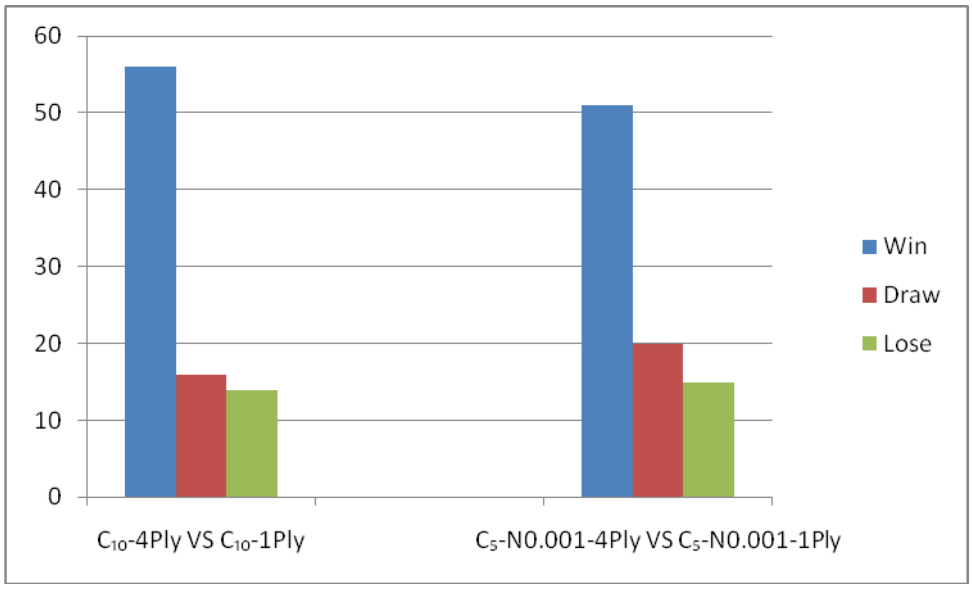

Figure 7.4 C_{10}-4Ply against C_{10}-1Ply and C_5-0.001-4Ply against C_5-0.001-1Ply.

Table 7.23 shows the mean and the standard deviation of the players' ratings after 5000 different orderings for the 86 played games, while table 7.24 summarises the results when playing the league between players using a starting position where all pieces are in their original positions (i.e. no two-move ballot).

	Mean	SD	Class
C_{10}-4Ply	1437.24	27.20	C
C_{10}-1Ply	1245.47	26.48	D
C5-0.001-4Ply	1354.60	27.43	D
C5-0.001-1Ply	1196.35	26.63	E

Table 7.23 Standard rating formula for all players after 5000 different orderings of the 86 games played.

		C_{10}-4Ply	C5-N0.001-4Ply
C_{10}-1Ply	Red	Lost	-
	White	Lost	-
C5-N0.001-1Ply	Red	-	Lost
	White	-	Lost

Table 7.24 Summary of Wins/Loses for C10-1Ply, C10-4Ply, C5-N0.001-1Ply and C5-N0.001-4Ply When not Using Two-Move Ballot.

The results in table 7.23 show that C_{10}-4Ply is significantly better than C_{10}-1Ply, and by using the average value for the standard rating formula, the results (when playing C_{10}-4Ply against C_{10}-1Ply) puts C_{10}-4Ply in class C (rating = 1437) and C_{10}-1Ply in class D (rating = 1245), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_{10}-1Ply and C_{10}-4Ply are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha. The results in table 7.23 also show that C_5-0.001-4Ply is significantly better than C_5-0.001-1Ply, and by using the average value for the standard rating formula, the results (when playing C_5-0.001-4Ply against C_5-0.001-1Ply) puts C_5-0.001-4Ply in class D (rating = 1354) and C_5-0.001-1Ply in class E (rating = 1196), and by using student t-test (assuming unequal variances, α = 0.05, and one-tail test), the results show that C_5-0.001-1Ply and C_5-0.001-4Ply are statistically different as the P value **(P-value=0**) for the one tail t-test is less than alpha.

7.8 SUMMARY

This chapter showed the importance of both the piece difference feature and the look-ahead depth to the game of checkers. Two main experiments

were carried out. The first one was to show the importance of the piece difference feature to evolutionary checkers by constructing many players. Sections 7.3 and 7.4 showed the experimental setup and the results.

The results showed that the piece difference is an important feature to the design of the evolutionary checkers based on Blondie24 and also to the design of the evolutionary checkers with round robin, individual and social learning and *n*-tuple systems.

The second experiment was intended to show the importance of a look-ahead depth to the evolutionary checkers. Sections 7.6 and 7.7 showed the experimental setup and the results. The results in sections 7.7.1, 7.7.2, 7.7.3 and 7.7.4 showed that the look-ahead depth is important to the design of the evolutionary checkers, also to the evolutionary checkers with round robin, individual and social learning and *n*-tuple.

An interesting point to note from the results is that increasing the depth level by one will give different performances depending on the level number, as the results indicates that increasing the level number from two to three gives a better performance than the performance gained when increasing the level number from one to two. The same occurs when increasing the depth level from three to four, which is better than increasing the depth from one to two and from two to three. According to this one can predict that the performance will increase when training at a level of five, six and so on. Also according to this increasing in the performance, there is no point of playing at a level that is lower than the trained level.

The results suggest that starting with a depth of four ply is the best value function to start with during learning phase for checkers. That is, train at four ply and then play at the highest ply possible.

In summary, the combination of piece difference feature with look-ahead depth seems a very important to the design of evolutionary checkers which worth applying to other computer games.

The next chapter will present the conclusions for all the work done in this thesis, together with some ideas for future work.

Chapter Eight

Conclusions and Future Work

8.1 CONCLUSIONS

The main focus of this thesis is in using evolution strategies to evolve neural networks to play checkers. As mentioned before, our objective was to propose a structure of learning methodologies for the game of checkers and also to produce a better player. We started by studying the background of evolution in game playing. Our work was inspired from Fogel's success in checkers in which his program, Blondie24 (Chellapilla and Fogel 2001; Fogel and Chellapilla 2002) was able to play a game of checkers at the human expert level, injecting as little expert knowledge as possible into the algorithm. We implemented a baseline player, C_0, which was based on the same architecture that Fogel used in the implementation of Blondie24. The objective was to investigate the performance of the evolved neural network player and also to obtain a baseline player that can be used to test the outcome of our proposed methods.

The objective of the first experiment was to eliminate the randomness in the choice of the opponents to play against during the evolution of C_0. We

implemented a round robin tournament, calling the resultant player Blondie24-RR. This player used exactly the same architecture that was used to evolve C_0, the only difference being the use of a round robin tournament, instead of randomly choosing opponents to play against. The results show that this small modification enhanced the player's ability to learn and hence produce a better player.

Our next experiment combined the ideas in (Kendall and Su 2003 and Su 2005). We evolved an individual and social checkers player in which a social pool was used to maintain the best player(s) at certain generations. Many experiments were carried out in order to test the outcome of increasing the number of best players in the social pool. The results were promising and encouraged us to also incorporate a round robin tournament within the individual and social learning framework. The resultant player, C_{10}-RR was the best evolved player in this thesis.

The success of *n*-tuple systems in many applications including optical character recognition, and evolving game playing strategies for the game of Othello provided the inspiration for us to investigate the *n*-tuple systems. Many sampling were considered and two main methods were used. The first method did not work well, suggesting that using just *n*-tuples with C_0 is not recommended. The second method showed that using *n*-tuples with TDL produced a good player using less computation time that required to evolve C_0. The experiments showed that using 5-tuples with random walk is the best sampling, among the selected samplings.

Finally all the experiments carried out in this thesis showed that the piece difference feature and the look-ahead depth were essential in evolving various checkers player.

With respect to further developing the methods used in this thesis, we recommend starting with an individual and social learning player that incorporates a round robin tournament together with a piece difference feature and which searches to a depth of at least four ply.

The research presented in this thesis has contributed in terms of learning techniques for evolutionary computer checkers. The main contributions are as follows.

1- In chapter three, we implemented an evolutionary checkers player, C_0. This player is based on the same architecture and structure that were used to construct Blondie24.

2- The results in chapter four demonstrated that it is possible to eliminate the randomness in choosing the opponents to play against during the evolution of C_0. The resultant player, Blondie24-RR was able to beat an online program and played well against two other strong programs, WinCheck3D and SXcheckers. The various players were compared to each other using the two-move ballot and the standard rating formula, which also confirmed that Blondie24-RR was superior to C_0.

3- The results in chapter five showed that introducing an individual and social learning method enhanced the learning process for the evolutionary checkers player and produced a superior player. The

resultant player, C_{10}, was better than C_0 and Blondie24-RR, shown using the two-move ballot and the standard rating formula.

4- The results in chapter five showed that increasing the number of players in the social pool will increase the performance for the evolved player. In this case values of 5 and 10 were the best values for determining when individual and social learning should occur. We implemented many players, where each player was constructed using a pair of values and all the players were played against each other to determine the best one.

5- The results in chapter five also showed that using a round robin tournament together with individual and social learning eliminated the randomness in the evolutionary phase of the resultant player, C_{10}-RR. We produced a superior player, which was better than C_0, Blondie24-RR and C_{10}, shown by the use of the two-move ballot and the standard rating formula.

6- The results in chapter six showed that using an *n*-tuples (with 5-tuple, constructed randomly, and with a 1-tuple) system, based on a Blondie24 architecture, produced a player that was worse than C_0. Therefore, it is not recommended to use only *n*-tuple systems for evolutionary checkers.

7- The results in chapter six also showed that using *n*-tuples (with 5-tuple constructed randomly) together with Temporal Difference Learning produced a player that was better than C_0 and Blondie24-RR. Evolving

an *n*-tuples player took only two days, which is much faster than the time required for C_0 and Blondie24-RR (19 days). This is because an *n*-tuples system is very fast and in TDL we only update the weights that are actually used. The evolved *n*-tuples player cannot beat C_{10} and C_{10}-RR but it was not our intention to produce a best player, rather we aimed to evolve a good player in a faster time than that required to evolve C_0, Blondie24-RR, C_{10} and C_{10}-RR. The *n*-tuples player, constructed using 5-tuple with random walks, was better than the *n*-tuples player constructed with 1-tuple. The experiments in chapter six also showed that the value of 0.001 is the best value for α among those tested values.

8- Considering all the players evolved in this thesis, C_{10}-RR (this player is based on the Blondie24 architecture and incorporates a round robin tournament and individual and social learning) was found to be the best overall player; with C_{10} being the second best and C_5-N0.001 being the third best. Blondie24-RR came forth and C_0 was the least successful. Table 8.1 summarises the results for all players.

		C_{10}-RR	C_{10}	C_5-N0.001	Blondie24-RR	C_0
C_{10}-RR	Red	-	Won	Won	Won	Won
	White	-	Won	Won	Won	Won
C_{10}	Red		-	Won	Won	Won
	White		-	Drawn	Won	Won
C_5-N0.001	Red			-	Won	Won
	White			-	Won	Won
Blondie 24-RR	Red				-	Won
	White				-	Drawn

Table 8.1 Summary of Wins/Loses for C_{10}-RR, C_{10}, C_5-N0.001, Blondie24-RR and C_0 when not Using the Standard Rating Formula.

9- The results in chapter seven showed that both the piece difference feature and look-ahead depth are very important in the design of an evolutionary checkers program, and is also important in the evolutionary checkers programs that used round robin, individual and social learning and *n*-tuple systems.

10- Using the results from the experiments in chapter seven we can conclude that a piece difference feature is important in the design of Blondie24. Of course, the neural network is also an important element of the whole design but the results presented here demonstrate that a simple feature is able to significantly improve the overall playing strength.

11- The experiments for showing the importance of a look-ahead depth that we have carried out in chapter seven produced many evolutionary checkers players, using different ply depths. Our expectations were that better value functions would be learned when training with deeper look-ahead search. This was found to be the case. The main results are that, during training and game playing,

better decisions are made when a deeper look-ahead is used. An interesting point to note is that increasing the depth level by one will give different performance depending on the level number. The results suggest that starting with a depth of four ply is the best value function to start with during learning phase for checkers. That is, train at four ply and then play at the highest ply possible.

8.2 FUTURE WORK

Based on the empirical investigations in this thesis, possible future works are as follows:

1- Apply the proposed methods that were developed in this thesis to other computer games such as Connect4. It will be interesting to see the outcome for the proposed methods in Connect4 since there are only two pieces in the game and there is no taking of the opponent's pieces, so there is no point of applying the methods with a piece difference feature. The Connect4 board consists of 42 squares, which means we need to find suitable neural network architecture for evolving the player and also to win the game. Since we need four pieces either in a row, column or diagonal to win the game, it is suitable to start the subsections of the Connect4 board from 4X4, and gradually increasing until the entire board is covered. This will change the number of weights of the input layer in the neural network architecture. This change in the number of weights may/may not require changing the number of hidden nodes in the

architecture; so many experiments are required to arrive at the best architecture for an evolved Connect4 player.

For an individual and social approach to Connect4, we need to test which values are suitable in deciding when the individual and social learning phases occur (i.e. test if increasing the number of players in the social pool helps evolve a better player or not).

When applying *n*-tuples to Connect4, since we have two values for the pieces then when using 5-tuples with a random walk, we only need (2^5=32) tuples, similarly for using 1-tuple, we only need (2^1=2) tuples, which means that evolving a Connect4 player using an *n*-tuple systems will much faster than evolving a checkers player using the same *n*-tuple system.

Finally, as we showed that the look-ahead depth is very important feature for the game of checkers, and this is not the case for many other games, it is really suggested to investigate if this feature is important for the game of Connect4 (or not). Our belief is that it is important because of the nature of the game, as to win the game we need four pieces in a row, column or in a diagonal and this requires look-ahead. We think that Connect4 needs to search for at least four ply depth but we want to make sure by actually evolve different Connect4 players, each one with different ply depths and testing the outcome for them, possibly by playing one another.

2- Investigate using individual and social learning for solving the problem of Blondie24 being an *end product*. This will bring about a continuous learning paradigm. One possible way to do this is by keeping the best player from each generation in the social pool and, in the case of an evolved player losing a match against a stronger player (human or computer), then we either use the second best player from the pool or randomly select one and test it (if possible) against the same player. This procedure will continue to run every time the evolved player loses against human or computer players. This will not guarantee a win for the evolved player, but at least we will have a player that is able to continuously change its strategy when losing, in a hope of a win.

3- When we carried out research into the *n*-tuple systems, we found that a key aspect is the right choice of the sampling and this is problem dependent. There are too many ways to sample a checkers board and testing all of them is a time consuming. Although we only test two of them and the results were promising; it is recommended to use many other *n*-tuple samples, for example using all the 3X3 subsections, 4x4 subsections, 3-tuple sampling or 4-tuple sampling, in order to arrive at a best *n*-tuple sample for the game of checkers.

4- Since most of the experiments in the thesis were constrained by run-time. We suggest enhancing the run-time by using the evolutionary enhancement aspects. Things like the parallel technology, which is used by (Franco et. al. 2010), could be useful to apply.

References

Al-Khateeb B. and Kendall G., Introducing a Round Robin Tournament into Blondie24, In Proceedings of the IEEE 2009 Symposium on Computational Intelligence and Games (CIG09), Milan, Italy, 2009, 112-116.

Al-Khateeb B. and Kendall G., The Importance of a Piece Difference Feature to Blondie24**,** In Proceedings of the the 10th Annual Workshop on Computational Intelligence (UK2010), Essex, UK, 2010, 1-6.

Al-Khateeb B. and Kendall G., Introducing Individual and Social Learning into Evolutionary Checkers, Transactions on Computational Intelligence and AI in Games (TCIAIG), (Under Review), 2011a.

Al-Khateeb B., Kendall G. and Lucas S., Introducing N-Tuple Systems into Evolutionary Checkers, Transactions on Computational Intelligence and AI in Games (TCIAIG), (Under Review), 2011b.

Al-Khateeb B. and Kendall G., The Importance of look ahead Depth in Evolutionary Checkers, In Proceeding of the 2011 IEEE Congress on Evolutionary Computation (CEC 2011), 2011c.

Anderson J. A. and Rosenfeld, E., Neurocomputing: Foundations of research, MA: MIT Press, 1988.

Axelrod R., The evolution of strategies in the iterated prisoner's dilemma, in L. Davis, ed, Genetic algorithms and simulated annealing, 1987, 32-41.

Bäck T. and Schwefel H. P., An overview of evolutionary algorithms for parameter optimisation, Evolutionary Computation, Vol. 1, 1993, 1-23.

Barto A.G., Sutton R.S. and Anderson C.W., Neuron like elements that can solve difficult learning control problems. IEEE Transactions on Systems, Man, and Cybernetics, Vol. 13, 1983, 834–846.

Baxter J., The evolution of learning algorithms for artificial neural networks, in D. Green and T. Bossomaier, eds., Complex Systems, Amsterdam: IOS, 1992, 313-326.

Baxter J., Tridgell A and Weaver L., Learning to play chess using temporal differences, Netherlands: Kluwer Academic Publishers, 2001.

Bettadapur P., and Marsland T.A., Accuracy and savings in depth-limited capture search, International Journal of Man-Machine Studies, Vol. 29, 1988, 497 – 502.

Bledsoe W. W., and Browning I., Pattern recognition and reading by machine, In *Proceedings of the Eastern Joint Computer Conference*, 1959, 225–232.

Burrow P. and Lucas S. M., Evolution versus temporal difference learning for learning to play Ms. Pac-Man, In Proceedings of the IEEE 2009 Symposium on Computational Intelligence and Games (CIG'09), Milan, Italy, 2009, 53-60.

Callan R. The essence of neural networks. Prentice Hall, 1999.

Campbell M., Joseph A., Hoane Jr. and Hsu F.-h., Deep Blue, Elsevier Science, 2002.

Campbell M., Knowledge discovery in Deep Blue, Communications of the ACM, Vol. 42, 1999, 65–67.

Carter M., Minds and computers: An introduction to the philosophy of artificial intelligence, Edinburgh University Press Ltd, 2007.

Caudell T. P. and Dolan C. P., Parametric connectivity: Training of constrained networks using genetic algorithms, in J. D. Schaffer, ed, proceedings of the 3rd internal conference on genetic algorithms and their applications, CA: Morgan Kaufmann, 1989, 370-374.

Chalmers D. J., The evolution of learning: An experiment in genetic connectionism, in proceedings of connectionist models summer school, 1990, 81-90.

Chellapilla K. and Fogel D. B., Evolving neural networks to play checkers without relying on expert knowledge. IEEE Transactions on Neural Networks, Vol. 10, 1999, 1382-1391.

Clark D., Deep thoughts on deep blue, IEEE Expert, Vol. 12, 1997, 31.

Coppin B., Artificial intelligence illuminated, MA: Jones and Bartlett Publishers, Inc., 2004.

Coulom R., Efficient selectivity and backup operators in monte-carlo tree search, In P. Ciancarini and H. J. van den Herik, editors, 5th International Conference on Computers and Games, Turin, Italy, 2006, 72-83.

Darwin C., On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life (5th ed.), London: John Murray, 1869.

Elman J. L., Finding structure in time, Cognitive Science, Vol. 14, 1990, 179-211.

Fang J. and Xi Y. G., Neural network design based on evolutionary programming, Artificial Intelligence in Engineering, Vol. 11, 1997, 155-161.

Fausett L., Fundamental of neural network: architectures, algorithms, and applications, NJ: Prentice-Hall International, Inc, 1994.

Fogel D. B., Evolving artificial intelligence, Ph. D. Thesis, UCSD, 1992.

Fogel D. B. and Chellapilla K., Verifying anaconda's expert rating by competing against Chinook: experiments in co-evolving a neural checkers player. Neurocomputing, Vol. 42, 2002, 69-86.

Fogel D. B., Blondie24 Playing at the Edge of AI, United States of America: Academic Press, 2002.

Fogel D. B., Hays T. J., Hahn S. L. and Quon J., A Self-Learning evolutionary chess program, in Proceeding of IEEE, IEEE Press, Vol. 92, 2004, 1947-1954.

Fogel D. B., Hays T. J., Hahn S. L. and Quon J., Further evolution of a self-learning chess program, In Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG05), Essex, UK, 2005, 73-77.

Fogel D. B., Hays T. J., Hahn S. L. and Quon J.: The Blondie25 Chess Program Competes Against Fritz 8.0 and a Human Chess Master. In Proceedings of the IEEE 2006 Symposium on Computational Intelligence and Games (CIG06), Reno, USA, 2006, 230-235.

Fogel D. B., Evolutionary Computation: Toward a new philosophy of machine intelligence (third edition), A John Wily & Sons, Inc., Publication, 2006.

Fogel D. B., Evolving a checkers player without relying on human experience. Intelligence, Vol. 11, 2000, 20-27.

Fogel D. B., An introduction to simulated evolutionary optimisation, IEEE Transactions on Neural Networks, Vol. 5, 1994, 3-14.

Fogel D. B., Using evolutionary programming to create neural networks that are capable of playing Tic-Tac-Toe, IEEE International Conference on Neural Networks. NJ: IEEE Press, 1993, 875-880.

Fogel D. B., An introduction to evolutionary computation, Australian Journal of Intelligent Information Processing Systems, Vol. 1, 1994, 34-42.

Fogel D. B., Evolutionary Computation: Toward a new philosophy of machine intelligence (Second edition). NJ: IEEE Press, 2000.

Fogel D. B., Wasson E. C. And Boughton E. M., Evolving neural networks for detecting breast cancer, Cancer Letters, Vol. 96, 1995, 49-53.

Fogel L. J., Owens A. J. and Walsh M. J., Artificial intelligence through simulated evolution, NY: John Wiley, 1966.

Franco M. A., Krasnogor N., and Bacardit J., Speeding up the evaluation of evolutionary learning systems using gpgpus. In the proceedings of the 12th annual conference on Genetic and evolutionary computation (GECCO '10), ACM, New York, USA, 2010, 1039-1046.

Galushkin A. I., Neural networks theory, Berlin Heidelberg: Springer-Verlag, 2007.

Garis, H., GenNets: Genetically programmed neural nets using the genetic algorithm to train neural nets whose inputs and/or output vary in time, IEEE International Conference on Neural Networks, 1991, 1391-1396.

Gelly S., Wang Y., Exploration exploitation in Go: UCT for Monte-Carlo Go, In: On-line Trading of Exploration and Exploitation, Whistler, BC, Canada, 2006.

Gelly S., Wang Y., Munos R. and Teytaud O., Modification of UCT with patterns in Monte-Carlo Go, 2006.

Goetsch G. and Campbell M.S., Experiments with the null-move heuristic, in T.A. Marsland, J. Schaeffer (Eds.), Computers, Chess, and Cognition, Berlin: Springer, 1990, 159–168.

Greenwood G. W., Training partially recurrent neural networks using evolution strategies, IEEE Transactions on Speech Audio Processing, Vol.5, 1997, 192-194.

Grönroos M., Whitley D. and Pyeatt L., A comparison of some methods for evolving neural networks, Genetic and Evolutionary Computation Conference, 1999, 1442-144.

Gruau F., Automatic definition of modular neural networks, Adaptive Behaviour, Vol.3, 1994, 151-183.

Hancock P. J. B., Genetic algorithms and permutation problems: A comparison of recombination operators for neural net structure specification, in D. Whitley and J. D. Schaffer, eds., International Workshop Combinations of Genetic Algorithms and Neural Networks, CA: IEEE Computer Society, 1992, 108-122.

Harley E., Book Review: Blondie 24, playing at the edge of AI, The IEEE Computational Intelligence Bulletin, Vol. 1, 2002, 25-27.

Harp S. A., Samad T. and Guha A., Designing application-specific neural networks using the genetic algorithm, in D. S. Tourezky, ed., Advances in neural information processing systems 2, CA: Morgan Kaufmann, 1990, 447-454.

Hart P.E., Nilsson N.J. and Raphael B., A Formal Basis for the Heuristic Determination of Minimum Cost Paths, *IEEE Transactions on Systems Science and Cybernetics,* Vol. 4, 1968, 100-107.

Haykin S., Neural networks: A comprehensive foundation (Second Edition), Pearson Education, Inc., 1999.

Heimes F., Zalesski G. Z., Land W. and Oshima M., Traditional and evolved dynamic neural networks for aircraft simulation, IEEE International Conference on Systems, Man and Cybernetics, 1997, 1995-2000.

Heinz E.A., Scalable search in computer chess, Friedrick Vieweg & Son, 2000.

Hopfield J. J., Neural networks and physical systems with emergent collective computational abilities, National Academy of Sciences, Vol. 79, 1982, 2554-2558.

Hoque S., Sirlantzis K., and Fairhurst M. C., "Bit plane decomposition and the scanning n-tuple classifier," In *Proceedings of International Workshop on Frontiers in Handwriting Recognition (IWFHR-8)*, 2002, 207–212.

Hsu F.-h, Anantharman T., Campbell M. and Nowatzyk A., A grandmaster chess machine, Scientific American, 1990, 44–50.

Hsu F.-h., Anantharman T.S., Campbell M.S. and Nowatzyk A., Deep thought, in: T.A. Marsland, J. Schaeffer (Eds.), Computers, Chess, and Cognition, Springer, Berlin, 1990, 55–78.

Hsu F.-h., Behind deep blue, NJ: Princeton University Press, Princeton, 2002.

Hsu F. h., IBM's deep blue chess grandmaster chips, IEEE Micro, Vol. 19, 1999, 70–81.

Hsu, F.-h., Large-scale parallelization of alpha-beta search: An algorithmic and architectural study, Ph.D. Thesis, Carnegie Mellon, Pittsburgh, PA, 1990.

Hughes E., Piece Difference: Simple to Evolve, The 2003 Congress on Evolutionary Computation (CEC 2003), Vol. 4, 2003, 2470–2473.

Igel C. and Stagge P., Graph isomorphisms affect structure optimization of neural networks, In International Joint Conference on Neural Networks, 2002, 142-147.

Junghanns A., Are there practical alternatives to alpha-beta in computer chess?, ICCA Journal. Vol. 21, 1998, 14–32.

Kaelbling L. P., Littman M. L. and Moore A.W., Reinforcement Learning: A Survey, Journal of Artificial Intelligence Research, Vol. 4, 1996, 237-285.

Kaindl H., Tree searching algorithms, in T. A. Marsland and J. Schaeffer, eds., Computers, Chess, and Cognition, NY: Springer-Verlag, 1990, 133-158.

Kanerva P., *Sparse Distributed Memory,* Cambridge, Mass.: MIT Press, 1988.

Kendall G. and Hingston P., Learning versus evolution in iterated prisoner's dilemma, Congress on Evolutionary Computation, 2004.

Kendall G. and Su Y., The co-evolution of trading strategies in a multiagent based simulated stock market through the integration of individual and social learning. In Proceedings of IEEE 2003 Congress on Evolutionary Computation, 2003, 2298–2305.

Kendall G. and Su Y. Learning with imperfections - a multi-agent neural genetic trading systems with different levels of social learning. In Proceedings of the IEEE Conference on Cybernetic and Intelligent Systems, 2004, 47–52.

Kendall G. and Su Y. Imperfect evolutionary systems. IEEE Transactions on Evolutionary Computation, Vol. 11, 2007, 294–307.

Kendall G. and Whitwell G., an evolutionary approach for the tuning of a chess evaluation function using population dynamics, Congress on Evolutionary Computation, 2001, 995-1002.

Kendall G., Yaakob R. and Hingston P., An investigation of an evolutionary approach to the opening of Go, Congress on Evolutionary Computation, 2004.

King D., Kasparov vs Deeper Blue: The ultimate man vs machine challenge. Badsford, 1997.

Kitano H., Designing neural networks using genetic algorithm with graph generation system, Complex Systems, Vol. 4, 1990, 461-476.

Kocsis L. and Szepesvari C., Bandit based monte-carlo planning. In 15th European Conference on Machine Learning (ECML), 2006, 282–293.

Kohonen T., Self-organizing maps, second edition. Berlin: Springer-Verlag, 1997.

Lee C., Wang M., Chaslot G., Hoock J., Rimmel A., Teytaud O., Tsai S., Hsu S. and Hong T., The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments, IEEE Transactions on Computational Intelligence and AI in Games (T-CIAIG), Vol. 1, 2009, 73-89.

Levene M., and Fenner T. I., "The effect of mobility on minimaxing of game trees with random leaf values," Artificial Intelligence, Vol. 130, 2001, 1-26.

Likothanassis S. D., Georgopoulos E. and Fotakis D., Optimizing the structure of neural networks using evolution techniques, 5th International Conference on Application of High-Performance Computers in Engineering, 1997, 157-168.

Louis S., Miles C., Playing to learn: Case-injected genetic algorithms for learning to play computer games, IEEE Transactions on Evolutionary Computation, Vol. 9, 2005, 669-681.

Lubberts A. and Miikkulainen R., Co-Evolving a Go-Playing neural network, In Coevolution: Turning adaptive algorithms upon themselves, Birds-of-a-Feather Workshop, Genetic and Evolutionary Computation Conference 2001.

Lucas S., The continuous n-tuple classifier and its application to real-time face recognition, In *IEEE Proceedings on Vision, Image and Signal Processing*, Vol. 145, 1998, 343–348.

Lucas S., Discriminative training of the scanning n-tuple classifier, in *Lecture Notes in Computer Science (2686): Computational Methods in NeuralModelling*. Berlin: Springer-Verlag, 2003, 222–229.

Lucas, S., Learning to Play Othello with N-Tuple Systems, Australian Journal of Intelligent Information Processing, Vol. 4, 2008, 1-20.

Lucas S. and Amiri A., Statistical syntactic methods for high performance OCR, In *IEEE Proceedings on Vision, Image and Signal Processing*, Vol. 143, 1996, 23–30.

Lucas S. and Cho K.T., Fast convolutional ocr with the scanning n-tuple grid, in *Proceedings of International Conference on Document Analysis and Recognition (ICDAR)*, IEEE Computer Society, 2005, 799–805.

Lucas S. and Kendall G., *Evolutionary Computation and Games,* IEEE Computational Intelligence Magazine, Vol. 1, 2006, 10-18.

Lucas S. and Runarsson T. P., Temporal difference learning versus coevolution for acquiring Othello position evaluation, In Proceedings of the IEEE 2006 Symposium on Computational Intelligence and Games (CIG'06), Reno Nevada, USA, 2006, 52-59.

Luger G. F., Artificial intelligence: Structures and strategies for complex problem solving (Fifth edition), Addison-Wesley, 2008.

McDonnell J. R. and Waagen D., Evolving recurrent perceptrons for time series modelling, IEEE Transactions on Neural Networks, Vol. 5, 1994, 24-38.

McDonnell J. R., Page W. and Waggen D., Neural network construction using evolutionary search, Third Annual Conference on Evolutionary Programming, 1994, 9-16.

Miconi T., Why Coevolution Doesn't "Work": Superiority and Progress in Coevolution, in L. Vanneschi, S. Gustafson, A. Moraglio, I De Falco, and M. Ebner (Eds.): Proceedings of the 12th European Conference on Genetic programming (EuroGP 2009), LNCS 5481, 2009, Springer.

Miikkulainen R, Evolving neural networks, GECCO (Companion) 2007, 3415-3434.

Miles C., Louis S., Cole N. and McDonnell J., Learning to play like a human: case injected genetic algorithms for strategic computer gaming, Congress on Evolutionary Computation (CEC2004), Vol. 2, 2004, 1441 – 1448.

Minsky M. L. and Papert S., Perceptrons, MA: MIT Press, 1969.

Mitchell M., An introduction to genetic algorithms, MIT Press, 1999.

Mitchell T. M., Machine learning, McGraw-Hill, 1997.

Moriarty D. E. and Miikkulainen R., Discovering complex othello strategies through evolutionary neural networks, Connection Science, Vol. 7, 1995, 195-209.

Moriarty D. E. and Miikkulainen R., Forming neural networks through efficient and adaptive co-evolution, Evolutionary Computation 1998, Vol. 5, 373-399.

Nasreddine H., Poh H. S. and Kendall G, Using an evolutionary algorithm for the tuning of a chess evaluation function based on a dynamic boundary strategy, IEEE Cybernetics and Intelligent Systems 2006, 1–6.

Nau D. S., Lustrek M., Parker A., Bratko I. and Gams M., "When Is It Better Not To Look Ahead?," Artificial Intelligence, Vol. 174, 2001, 1323-1338.

Newborn M., Kasparov vs. Deep Blue, Computer Chess Comes of Age. New York: Springer-Verlag, 1997.

Nilsson N. J., Artificial Intelligence, A new Synthesis. Morgan Kaufmann, 1998.

Norvig P., Paradigms of artificial intelligence programming, Morgan-Kaufmann, 1992.

Park J. and Sandberg J. W., Universal approximation using radial basis functions network, Neural Computation, Vol. 3, 1991, 246-257.

Patterson D.W., Artificial Neural, Networks theory and applications, Prentice Hall, 1996.

Pollack J. B. and Blair A. D., Co-evolution in the successful learning of backgammon strategy, Machine Learning, Vol. 32, 1998, 225-240.

Porto V. W., Fogel D. B. And Fogel L. J., Alternative neural network training methods, IEEE Expert, Vol. 10, 1995, 16-22.

Rechenberg I., Cybernetic solution path of an experimental problem, Royal Aircraft Establishment, August, 1965.

Reynolds R. G., An Adaptive computer model of the evolution of agriculture for hunter-gatherers in the Valley of Oaxaca, Ph.D. dissertation, Univ. Michigan, Ann Arbor, MI, 1979.

Reynolds R. G, An introduction to cultural algorithms, in *Proc. 3rd Annu.Conf. Evol. Program.* 1994, 131–139.

Reynolds R. G., Kobti Z., and Kohler T. A., The effects of generalized reciprocal exchange on the resilience of social networks: An example from the prehistoric Mesa Verde region, *J. Comput. Math. Organ. Theory*, Vol. 9, no. 3, 2003, 229–254.

Reynolds R. G., Kobti Z., Kohler T. A., and Yap L., Unravelling ancient mysteries: Reimagining the past using evolutionary computation in a complex gaming environment, *IEEE Trans. Evol. Comput.*, Vol. 9, no. 6, 2005, 707–720.

Reynolds R. G. and Peng B., Cultural algorithms: Knowledge learning in dynamic environments, in *Proc. IEEE Int. Congr. Evol. Comput.*, 2004, 1751–1758.

Richards N., Moriarty D. E. and Miikkulainen R., Evolving Neural Networks to Play Go, Applied Intelligence, Vol. 8, 1998, 85-96.

Robertie B., Carbon versus silicon: Matching wits with TD-Gammon, Inside Backgammon, Vol. 2, 1992, 14–22.

Rohwer R., and Morciniec M., A theoretical and experimental account of n-tuple classifier performance, *Neural Computation*, Vol. 8, 1996, 629 – 642.

Rohwer R., and Morciniec M., The Theoretical and Experimental Status of the *n*-tuple Classifier, *Neural Networks*, Vol. 11(1), 1998, 1–14.

Rosin C.D. and Belew R.K., New methods for competitive coevolution. Evolutionary Computation, Vol. 5, 1997, 1–29.

Rosenblatt F., The Perceptron: A Probabilistic model for information storage and organization in the brain, Psychological Review, Vol.65, 1959, 386-408.

Rosenblatt F., Principles of neurodynamics, NY: Spartan Books, 1962.

Runarsson T.P. and Jonsson E.O, Effect of look-ahead search depth in learning position evaluation functions for Othello using ε–greedy exploration**,** In Proceedings of the IEEE 2007 Symposium on Computational Intelligence and Games (CIG'07), Honolulu, Hawaii, 2007, 210 - 215.

Runarsson T.P., and Lucas S.M., Coevolution versus self-play temporal difference learning for acquiring position evaluation in small-board go, IEEE Transactions on Evolutionary Computation, Vol. 9(6), 2005, 628–640.

Russell S. and Norvig P., Artificial intelligence: a modern approach (Third edition), Prentice Hall, 2010.

Samuel A. L., Some studies in machine learning using the game of checkers, IBM Journal on Research and Development, 1959, 210-229. Reprinted in: E. A. Feigenbaum and J. Feldman, eds., Computer and Thought, NY: McGraw-Hill, 1963. Reprinted in: IBM Journal on Research and Development, 2000, 207-226.

Samuel A. L., Some studies in machine learning using the game of checkers, IBM Journal of Research and Development, 1959, 210-229.

Sarkar M. and Yegnanarayana B., Evolutionary programming-based probabilistic neural networks construction technique, IEEE International Conference of Neural Networks, 1997, 456-461.

Schaeffer J., One jump ahead: Computer Perfection at Checkers. New York: Springer, 2009.

Schaeffer J., Burch N., Björnsson Y., Kishimoto A., Müller M., Lake R., Lu P and Sutphen S., Checkers Is Solved, *Science* Express, Vol. 317, 2007, 1518 – 1522.

Schaeffer J., Joseph C. and Norman T. A., World championship caliber checkers program, Artificial Intelligence, Vol. 53, 1992, 273-290.

Schaeffer J., Lake R. and Lu P., Chinook the world man-machine checkers champion, AI Magazine, Vol. 17, 1996, 21-30.

Schaeffer J., Treloar N., Lu P. and Lake R., Man versus machine for the world checkers championship, AI Magazine, Vol. 14, 1993, 28-35.

Schlang M., Poppe T. and Gramckow O., Neural networks for steel manufacturing, IEEE Expert Intelligent Systems, Vol. 11, 1996, 8-10.

Schwefel H. P., Numerical optimization of computer models, Chichester, U.K.: Wiley, 1981.

Simon H. A., *Models of bounded rationality: Empirically grounded economic reason*, volume 3. MIT Press, Cambridge, MA, 1997.

Simon H. A., *Administrative Behavior, Fourth Edition*. New York, NY: The Free Press, 1997.

Smet P., Calbert G., Scholz J., Gossink D., Kwok H-W, and Webb M., The Effects of Material, Tempo and Search Depth on Win-Loss Ratios in Chess, A1 2003: Advances in artificial intelligence, Lecture Notes in Computer Science, Vol. 2903 , 2003, 501-510.

Specht D.F., Probabilistic neural networks, Neural Networks, Vol. 3, 1990, 110-118.

Stanley K., Exploiting Regularity Without Development, In Proceedings of the AAAI Fall Symposium on Developmental Systems, Menlo Park, CA, 2006, AAAI Press.

Stanley K. and Miikkulainen R., Evolving neural networks through augmenting topologies, IEEE Transactions on Evolutionary Computation, Vol. 10, 2002, 99-127.

Stanley K., Cornelius R. and Miikkulainen R, Real-time learning in the NERO video game, AIIDE 2005, 2005, 159-160.

Sternberg M. and Reynolds R. G., Using cultural algorithms to support re-engineering of rule-based expert systems in dynamic environments: A case study in fraud detection, *IEEE Trans. Evol. Comput.*, Vol. 1, no. 4, 1997, 225–243.

Sutton R.S., Temporal Credit Assignment in Reinforcement Learning., PhD thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, 1984.

Sutton R. S., Two problems with Backpropagation and other steepest-descent learning procedures for networks, 8th Annual Conference of Cognitive Science Society. NJ: Hillsdale, 1986, 823-831.

Sutton R. S. Learning to predict by the methods of temporal differences. Machine Learning, Vol.3, 1988, 9–44.

Tesauro G., Neurogammon wins computer olympiad, Neural Computation Vol. 1**,** 1989, 321-323.

Tesauro G., Neurogammon: A Neural-Network Backgammon Program, IJCNN International Joint Conference on Neural Networks, 1989, 33-39.

Tesauro G., Practical issues in temporal difference learning, Machine Learning, Vol.8, 1992, 257–277.

Tesauro G., Temporal difference learning and TD-Gammon, Comm. ACM, Vol. 38, 1995, 58–68.

Tesauro G., Programming backgammon using self-teaching neural nets, Artificial Intelligence, Vol. 134, 2002, 181–199.

Tesauro G. and Sejnowski T. J., A parallel network that learns to play backgammon, Artificial Intelligence, Vol. 39, 1989, 357-390.

Turing A. M., Computing machinery and intelligence, Mind, Vol.59, 1950, 433-460.

Ullman J., Experiments with the n-tuple method of pattern recognition, IEEE *Transactions on Computers*, Vol. 18(12), 1969, 1135–1137.

Vriend N., An illustration of the essential difference between individual and social learning, and its consequences for computational analysis, Journal of Economic Dynamics and Control,Vol. 24,2000, 1-19.

Vrakas D., Vlahavas I. PL., Artificial intelligence for advanced problem solving techniques, Hershey, New York, 2008.

Whitely G. and Ligomenides P., GANet: A genetic algorithm for optimizing topology and weights in neural network design, International Workshop on Artificial Neural Networks, 1993, 322-327.

Werbos P.J., The Roots of Backpropagation: From ordered derivatives to. Neural Neworks and Political Forecasting, John Wiley and Sons, New York, 1994.

Whitley D., Starkweather T. and Bogart C., Genetic algorithms and neural networks: Optimizing connections and connectivity, Parallel Computation, Vol. 14, 1990, 347-361.

Whitley D., Ranaa S., Dzubera J. and Mathias K. E., Evaluating evolutionary algorithms, Elsevier B.V., Artificial Intelligence, Vol. 85, 1996, 245-276.

Williams R. and Zipser D., Gradient-based learning algorithms for recurrent networks and their computational complexity, in Y. Chauvin and D. Rumelhart, eds., Backpropagation: Theory, architectures, and applications. NJ: Lawrence Erlbaum Associates, 1995, 433-486.

Yao S., Wei C. J. and He Z. Y., Evolving wavelet neural networks for function approximation, Electronic Letters, Vol. 32, 1996, 360-361.

Yao X., Evolving artificial neural networks, in Proceedings of the IEEE, Vol. 87, September 1999a, 1423-1447.

Yao X., The importance of maintaining behavioural link between parents and offspring, IEEE Conference on Evolutionary Computation, 1997, 629-633.

Yao X. and Liu Y., EPNet for chaotic time series prediction, in X., Yao; J. H. Kim and T. Furuhashi, eds., Selected Papers of 1st Asia-Pacific Conference on Simulated Evolution and Learning, Lecture Notes in Artificial Intelligence, Berlin: Springer-Verlag, 1997b, 146-156.

Yao X., Evolutionary computation – Theory and applications, World Scientific Publishing, 1999b.

Yao X. and Islam Md. M., Evolving artificial neural network ensembles, IEEE Computational Intelligence Magazine, Vol. 3, 2008, 31-42.

Zirilli J. S., Financial prediction using neural networks, MA: International Thomson Publishing, 1996.

Zurada J. M., Introduction to artificial neural systems, JAICO publishing House, 1996.