Using Particle Swarm Optimization to Evolve Two-Player Game Agents

Leon Messerschmidt

Submitted in partial fulfilment of the requirements for the degree Magister Scientiae

in the Faculty of Engineering, Built Environment and Information Technology

University of Pretoria

Pretoria, South Africa

2005

Using Particle Swarm Optimization to Evolve Two-Player Game Agents

Leon Messerschmidt

2005

Abstract

Computer game-playing agents are almost as old as computers themselves, and people have been developing agents since the 1950's. Unfortunately the techniques for game-playing agents have remained basically the same for almost half a century – an eternity in computer time. Recently developed approaches have shown that it is possible to develop game playing agents with the help of learning algorithms. This study is based on the concept of algorithms that learn how to play board games from zero initial knowledge about playing strategies.

A coevolutionary approach, where a neural network is used to assess desirability of leaf nodes in a game tree, and evolutionary algorithms are used to train neural networks in competition, is overviewed. This thesis then presents an alternative approach in which particle swarm optimization (PSO) is used to train the neural networks. Different variations of the PSO are implemented and compared. The results of the PSO approaches are also compared with that of an evolutionary programming approach. The performance of the PSO algorithms is investigated for different values of the PSO control parameters. This study shows that the PSO approach can be applied successfully to train game-playing agents.

Thesis supervisor: Prof. A.P. Engelbrecht

Department of Computer Science

University of Pretoria

Acknowledgments

Special thanks go to these people who have helped in the production of this thesis:

- *A.P. Engelbrecht*, for his advice and insight into the sometimes dark inner workings of learning algorithms.
- *R. Messerschmidt*, for his valuable advice on the statistical analysis. He is also my playing partner.
- *N. Messerschmidt*, for patiently waiting for her husband, while I was working on this study. Without her support, encouragement and patience this study would not have seen the light.
- *University of Pretoria, IT Department*, for allowing the use of their computer labs over weekends and late nights.

"The question of whether a computer can think is no more interesting than the question of whether a submarine can swim."

-Edsger W. Dijkstra

Contents

List of Figures

List of Tables

Chapter 1

Introduction

From the very early days of their existence, humans have played games for relaxation and for measuring their skills against each other. Many games test physical skill and stamina, for example tennis or football. Other games, such as checkers, chess, or go test the mental prowess and ingenuity of the players. Some games contain elements of both categories.

This study concentrates on mental or thinking games, in which players must use careful thought to weigh different options. These thinking games are often in the category of board games, i.e, games that are played with tokens or pieces on a two-dimensional board.

Board games are often simplified versions of real-world events or problems. For example, chess and go are said to have their origins in the tactical and strategic concepts of warring armies [64][67]. Monopoly is a game of economic strategy.

Board games are matches of wit or intelligence between humans, requiring strategic thinking, as well as creativity and ingenuity to outwit an opponent. The ability to win frequently in these games is often associated with intelligence. In western societies, individuals will sometimes attempt to prove superior intellect by challenging others to a game of chess. In eastern societies, an accomplished skill in go is said to come from patience and wisdom – so much so that some proverbs can find their roots in go.

That games are deeply rooted in human behavior is hardly surprising, for a game is an

outlet to the competitive nature of humans. Humans like to compete against each other, or against other creatures and even machines! Sometimes humans win and sometimes they lose, but they will always compete.

Even so, machines have surpassed us in many physical and computational abilities. The ability of intelligent thought has, however, always been seen as the hallmark of the human species. Our ability to reason gives us the edge above any other living form. In the arena of creative thinking, the human race remains unchallenged by any animal or machine.

Games that require thought, wit, and creativity are seen as the ultimate match between human and machine. There is a psychological reason for attempting to create a computer player that will beat any human. It is the attempt to dethrone humans as undisputed kings in the area of intelligent game play.

Games that require intelligent thought are usually two-player, zero sum, perfect knowledge, turn-based board games. In two-player games only two-players take part at a time. Zero sum indicates games where the amount won by player will equate to the amount lost by the other player. Perfect knowledge means that both players have all available information and that there are no hidden features or elements of chance, such as dice or hidden cards. In turn-based games, players have mutually exclusive turns to make a move. Finally, board games are played on a board (usually two-dimensional) where the current state of the game is shown.

This study is about using computers to create game-playing agents that can play two-player, zero sum, perfect knowledge, turn-based board games. Artificial intelligence techniques, as well as some more traditional techniques, are tested to establish good ways to create game playing agents. This study is at its core an attempt to improve some on earlier game-playing algorithms.

There are numerous reasons for studying, creating, and improving game-playing agents. Firstly, computer game programs provide human players with an opportunity to practice against a tireless opponent, an opponent that will be available any time of the day or night.

Secondly, games provide a unique opportunity to apply computational intelligence tech-

niques in a known problem space. Because games provide a predefined complex environment (the board and game rules) and a well-defined goal, it is possible to measure the performance of different algorithms and to compare the effectiveness of various computational intelligence techniques [15][86] against each other, or against humans, to measure the actual play strength of an agent.

Although games are relatively simple abstractions, they are often too complex to solve in a reasonable amount of time. In other words, it is impractical to write a simple algorithm that will play perfect chess. Chess is too complex for simple algorithms and a more sophisticated approach is needed, where a computer agent needs to plan ahead based on limited knowledge. However, board games are often simple enough so that students of computational intelligence can understand the problem domain relatively easily [86]. Researchers in computational intelligence can therefore concentrate on the actual operation of agents and not invest time defining the problem space.

Thirdly, the opportunity to match one's programming skill against that of a professional human game master is often too much to resist for many computer scientists.

This study focuses on the second reason for developing game agents: the application of modern computational intelligence technologies to game domains.

There are many computational intelligence techniques that are available to computer scientists to solve a wide range of problems. Many of these techniques can be applied to a wide range of different problems and disciplines [24] [66] [73]. Unfortunately, it is not always clear which technique to use under specific circumstances. The effectiveness of new techniques or new combinations of existing techniques may also not be tested or well known [24].

Some computational intelligence techniques are variations on existing technology, for example, a variant of particle swarm optimization [22], called guaranteed convergence particle swarm optimization (GCPSO)[96]. Although these strategies have been shown to find optimal solutions in controlled conditions, they have seldom been applied in a real-life environment. This study provides a testing ground for some new techniques to game-playing agents, specif-

ically PSO algorithms.

The most successful game playing algorithms rely on hand-crafted algorithms that are painstakingly put together by humans. In contrast with traditional algorithms (for example, Chinook [84] and Deep Blue [47]), computational intelligence algorithms rely on the ability to learn and adapt to features of a game, in other words, to learn play strategies from experience and observation. Existing techniques are almost always rule-based – essentially boiling down to a series of if-then-else rules that have to be created by humans. This requires very long and tiresome hours – so much so that creators of good game agents have publicly admitted that the process was less than enjoyable [84]. A point has been reached where a better way of creating game agents is needed.

Around the turn of the century, Chellapilla and Fogel produced a "self tough" artificial intelligence game agent [10]. They successfully created a game playing-agent, based on a neural network – trained using coevolutionary algorithms – that managed to learn the game of checkers without any knowledge of game strategy and planning. The game agent (called Blondie24) started to learn the game of checkers with no more knowledge than the position, game tree and number of pieces and rules of the game. The amazing thing is that Blondie24 is able to beat most human opponents and even the odd master player [27]!

Artificial intelligence agents may even surpass the abilities of traditional algorithms in the near future. The goal of this study is to develop a new artificial intelligence game-playing agent, and more specifically to:

1. Develop an effective measuring criterion for game-playing agents. The measuring criterion must be automated, objective, and repeatable.

2. Show that particle swarm optimization (PSO) can be applied to training a game-playing agent for tic-tac-toe.

3. Show that PSO can be applied to training a neural network that can be used as a checkers evaluation function.

4. Study the effects of control parameters of PSO on the performance of game playing-agents.

5. Evaluate the performance of PSO game-playing agents, and to compare these results with other strategies.

6. To develop or use techniques that are generally applicable to most board games, i.e, not to rely on game-specific implementations.

Chapter 2 gives an overview of the current state of game-playing technology. Chapter 3 covers the general game-playing framework used in this study. This framework can be applied to most board games. Chapters 4 and 5 cover specific implementations based on the framework, shown in chapter 3, as well as the results of game-playing agents for tic-tac-toe and checkers, respectively. These chapters summarize the implementation of evolutionary game algorithms and introduce a new PSO-based technique for training game-playing agents. Chapter 6 concludes the study with an overview of the results and further areas of study.

Chapter 2

Background

This chapter reviews and evaluates the background of computer game-playing agents. It consists of summaries of traditional approaches such as tree-based searches and also covers computational intelligence gaming techniques. The computational intelligence algorithms that are reviewed include neural networks, evolutionary computation, and particle swarm optimization strategies.

2.1 Introduction

Since the early days of computing the ability of computer scientists to create algorithms that can play against (and even beat) human opponents in board games has fascinated computer scientists [90]. The ability of computers to compete against humans, provides a tangible showcase of what computers of the day can do [27][47]. For example, a specialized IBM computer named Deep Blue could explore and evaluate two million chess moves in a second [47], and off-the-shelf hardware can play a great game of competitive checkers [84]. Game-playing agents give laymen a unique window into the power of modern computing.

Because of this fascination and almost constant attention, a range of different techniques for game-playing agents have been developed in almost five decades. These techniques have

been governed by the constraints of computer hardware and software techniques at the time. For example, the chess rating of chess playing computer agents shows a linear correlation with processor speed [27]. Game agents often reflect the development of hardware and software technologies over time – starting from primitive tree searches and simple evaluation functions in the 1950's and 1960's [87][90] to multi-processor hardware and parallel tree searches [47][84] in the 1990's.

As Moore's Law (i.e, processor speed doubles roughly every 18 months) continues to drive computers faster [78], these techniques will continue to be developed and refined, making it possible to implement computer agents for even the most complex games. Some of these games have convoluted search spaces and extremely difficult evaluation functions.

Although modern computer hardware have enabled computer scientists to create spectacular game playing agents, the search for better game playing agents has not reached an end. For some games, computer algorithms feature as some of the strongest players in the world [84]. For others, computer players are not strong enough to provide strong resistance to even amateur players [6].

The search space (or all possible board states in a game) for most games is too large to be computed in a reasonable amount of time, nor is it possible to store these data on any modern storage device. Game-playing agents therefore have to plan ahead and generate a good move with relatively little information available about the search space.

Improving the game-playing performance of computer players can be done in two ways: A game agent can attempt to search a larger part of the search space, or the agent can attempt to make better use of the information that is available (or both). Although the first attempt does work, a turning point is reached where bigger portions of the search space deliver diminishing returns - each layer of additional information is less valuable than the previous layer for checkers [84]. It is therefore more effective to follow the second route, which is the main objective of this study.

This chapter introduces concepts of game-playing agents. Section 2.2 defines the concept

of solving a game, while sections 2.3 and 2.4 cover common techniques for creating game-playing agents without exhaustive knowledge. Sections 2.5, 2.6, and 2.7 give an overview of training techniques for artificial intelligence agents. The rest of the chapter covers some of the successful game-playing agents that are based on the algorithms covered in this chapter.

2.2 Solving a Game

A solution to a game is a sequence of rules, or actions, that will enable a human player or computer program to force a win at best, or a draw at worst. A good solution consists of a concise set of rules, usually determined from a game tree. This section presents a concept solution for the simple game of tic-tac-toe.

For games with a small search space, for example, tic-tac-toe, it is possible to find a complete solution. However, most modern games have a large, convoluted search space, which makes it impossible to construct a complete search tree due to current hardware constraints. It is therefore not always possible to find a solution to a game.

A solution for tic-tac-toe as used for this study is as follows:

1. If a winning state can be achieved, make the winning move.
2. If the opponent can win in the next move, block him.
3. Test for the exception to the rule (refer to figure 2.1).
4. If the middle block is open, move in the middle block.
5. If a corner block is open, move to any corner block.
6. Move to any side block.

Figure 2.1 illustrates the exception to the rule mentioned in *rule 3*, where the computer agent is playing as "O" and the opponent as "X", and "O" has to make the next move. If the

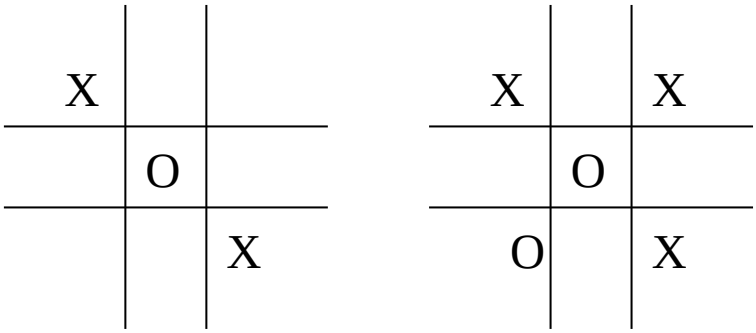

Figure 2.1: Exception to the tic-tac-toe Rule

board position on the left hand side of figure 2.1 is the current position, *rule 5* states that the move should be made in a corner position. A corner position will cause the opponent to reach a position where he can force a win (refer to the right hand side of figure 2.1). In this case it is necessary to move to a side block to avoid a forced win.

Any game-playing agent with actions that are based on rules that solve a game will be unbeatable. Furthermore, the agent will be very efficient in terms of the time and resources taken to compute moves. Even though most modern games cannot be solved at this stage, solving games will always be a good goal to strive for. Some of the techniques discussed in this study may lay the ground-rules to solve games with artificial intelligence techniques.

The rest of this chapter discusses some popular techniques to compute good moves without solving the game. Some existing game-playing agents are also discussed.

2.3 Game Dictionaries

A simple, yet very effective, way of creating a game agent using artificial intelligence is to create a stimulus-response map [73]. A stimulus-response map is basically a dictionary-type lookup table. The map gives, for each possible game state, at least one move that is known to be a good move (a move that leads to winning). The stimulus-response map is often called a game dictionary.

To find an action to execute, the game agent observes the current board state, and performs a lookup for this state in the game dictionary. The agent then responds with any of the known actions listed in the game dictionary. Because the actions in the game dictionary are known to be good actions, the game agent makes strong moves. The ultimate strength of the game agent is directly dependent on the quality of the game dictionary. A game dictionary strategy is relatively simple and easy to implement.

Game dictionaries will not always force a winning position. The responses listed in a dictionary may only be known strong moves, but not necessarily the best moves. Note that a complete game dictionary that constantly forces a win (or at least a draw) for a player is one way to solve a game. However, to create a game dictionary it must be possible to create and store the entire game tree. This is unfortunately only possible with very simple games like tic-tac-toe. Larger games like chess, which is estimated to have upward of 10^{44} possible positions [91], cannot be solved with game dictionaries. Even with state-of-the-art hardware it is not possible to compute all the possible moves of these games. It is also not yet possible to store the enormous quantities of data contained in the entire game tree, as shown by the US$10000 bounty that is still out on www.arimaa.com for the game of arimaa [92].

Even though it is not possible to create a complete game dictionary, most successful game-playing agents make use of a partial game dictionary for key parts of the game, for example, Deep Blue's chess opening dictionary [47] or Chinook's end-game dictionary [84]. Game agents that use partial game dictionaries rely on the dictionaries to provide the best moves at key points in the game, where it is difficult to compute moves with a partial game tree.

For the rest of the game, dynamic planning systems are used to compute strong moves. This thesis deals with dynamic planning systems based on computational intelligence techniques.

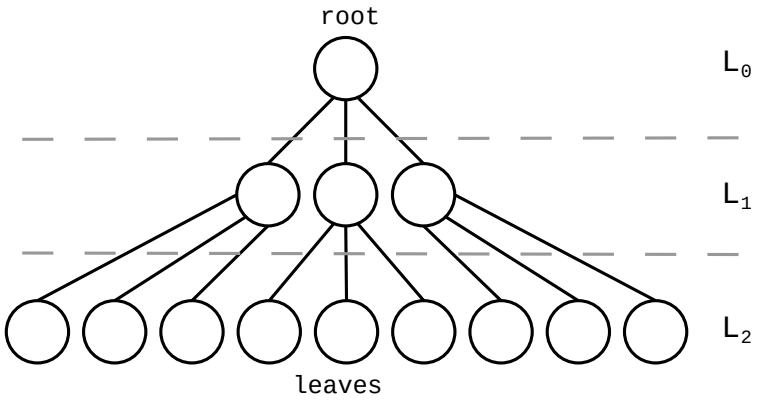

Figure 2.2: Game Tree

2.4 Tree-Based Algorithms

This section covers the development of game playing agents with the help of tree-based algorithms. It discusses the different methods of creating a tree-based agent and thereafter discusses the limits of these algorithms.

Because of the large number of possible moves of most two-player games (including checkers, chess, and go), it is not possible to compute all possible moves. This makes it impossible to solve the game with a dictionary style agent. Agents that have to compute moves for these games use a planning algorithm that enables them to compute a move without having perfect or complete information.

The most popular planning algorithms for game-playing agents are based on tree algorithms. A tree algorithm is used to compute a number of future moves, although computing and storage limitations do not allow for the computation of all possible future moves for most games.

2.4.1 Constructing the Game Tree

To build a game tree, the tree building algorithm starts at the current board state. With reference to figure 2.2, let L_0 denote the initial board state. The next tree level, L_1, is

constructed by creating a node for each legal move from L_0. To create the next tree level (level L_2) each legal move from each node in level L_1 is used to create a new node in level L_2. This process continues until one of the following happens:

1. a predefined ply-depth has been reached,
2. the complete tree has been constructed, or
3. resources such as storage or computation time have been depleted.

The first level, containing the initial state, is called the root node. The last level of the tree contains nodes that do not contain any child nodes. These nodes are called leaf nodes. In figure 2.2, L_0 is the root node and all the nodes in level L_2 are leaf nodes.

Generally, a game agent plays better with deeper trees (trees with more levels). However, at very deep levels play strength will only increase marginally for checkers players [84].

It is possible for a game tree to contain duplicate nodes, for there are often more than one way to reach a game state. To improve the performance of the tree building algorithm, such duplicate nodes can be removed; when a new node is constructed the existing tree is examined for an identical node. In the case of two or more duplicate nodes, only one node is expanded [73]. Removal of duplicate nodes saves memory and reduces computational complexity, since the same nodes will not be expanded twice.

Once the partial (or sometimes complete) game tree is constructed, it is used to plan ahead and compute the next move. Several game tree construction and planning algorithms have been developed, of which the minimax and alpha-beta algorithms are used widely.

2.4.2 Minimax Search

Minimax search is a simple, yet relatively effective, tree-based planning algorithm. Minimax is based on a tree building process for games that are played by two opponents, called Min and Max [62].

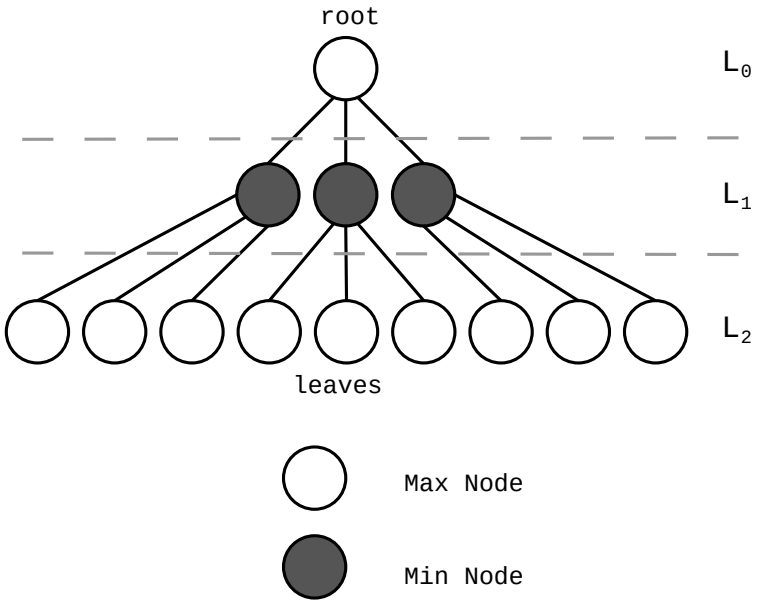

Figure 2.3: Minimax Game Tree

The algorithm assumes that the current state, or root node of the tree, is the position where it is the Max player's turn to move (refer to figure 2.3). To generate the next layer, all the legal moves for Max are computed. Thereafter, the next layer contains all the possible moves for the Min player. The tree building process alternately creates layers for each player until one of the termination criteria, mentioned in the previous section, is met.

A node in a Max layer is referred to as a Max node, while nodes in a Min layer are referred to as Min nodes.

After construction of the minimax tree, the leaf nodes are evaluated with a static evaluation function. The static evaluation function quantifies the strength or desirability of the board state represented by the leaf node. For minimax to work, the evaluation function should evaluate to larger values for the current player (Max) and smaller values for the opponent (Min). Multiplying the function by -1 can reverse the Max player's function to apply to the Min player.

After each leaf node has been evaluated, the values are "sent back" up the tree (refer to

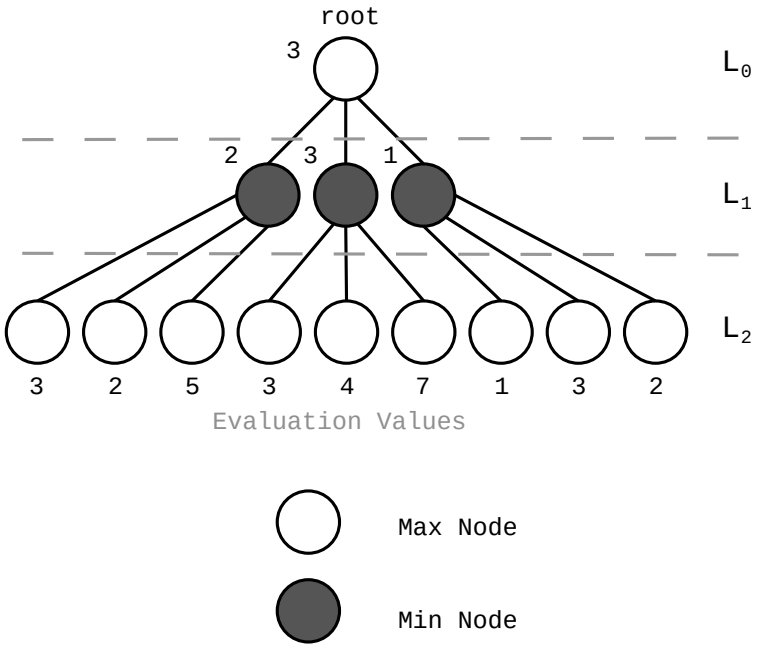

Figure 2.4: Minimax Game Tree With Evaluation Values

figure 2.4). Each node receives the best evaluation function of all its child nodes. It is assumed that the game agent always attempts to make the best possible move. The max player selects the child node with the highest evaluation value. On the other hand, it is assumed that the opponent (Min player) always attempts to make the move that is the worst for the Max player. The opponent therefore chooses the node that has the worst evaluation value. Because of the alternating levels of min and max nodes, minimax alternately chooses the best and worst evaluation value until the root node is reached [73].

The evaluation value propagated to the root node represents the move that yields the best possible position for as far as the tree has been constructed [73].

The minimax algorithm has a number of limitations:

- Parts of the minimax tree may not represent any useful board states. The algorithm therefore wastes time and memory on evaluating states that will never be considered to be viable moves.

- For each new move, the entire tree is rebuilt from the current board state. Because the tree is discarded when the move is made, a large amount of computational and effort go to waste.

- Because minimax assumes the opponent will make the best move according to the static evaluation function, some positions are not evaluated correctly. For example, sacrificing a piece in chess to gain a strategic advantage will fool evaluation functions that concentrate on a material advantage.

2.4.3 Alpha-Beta Search

A big computational drawback of the minimax search is that it has to evaluate a large number of nodes that do not lead to promising states. If moves that do not lead to good board states can be identified earlier in the tree building process, a large amount of computation time and storage can be saved. The alpha-beta algorithm provides an alternative to address these problems of the minimax algorithm [62].

In contrast with minimax, which generates the entire tree before leaf nodes are evaluated, the alpha-beta search evaluates leaf nodes as soon as they are created. This early evaluation is possible because alpha-beta uses a depth-first search as opposed to a breadth-first search [73]. Once a node has been evaluated, its evaluation value is used to determine whether other parts of the tree can be omitted.

The basic algorithm for alpha-beta consists of global minimum and maximum values called Alpha and Beta respectively. Alpha holds the best evaluation value reached in the entire tree building process, while Beta holds the worst evaluation value. Nodes with evaluations stronger than Alpha or weaker than Beta will not be expanded. See the following pseudo algorithm for more detail:

$alphabeta(n, alpha, beta)$

1. If the depth bound has been reached, return the static evaluation function for node n

2. Generate the child nodes for node n

3. For each child node c

   (a) Set x to the return value of alphabeta (c,-alpha,-beta)

   (b) If x is greater or equal to beta, return x

   (c) If x is greater than alpha, set alpha $= x$

4. Return alpha

Note that alpha-beta yields the same result as minimax, but usually the time and memory used to reach the result are reduced [73]. In the worst case alpha-beta is equivalent to minimax. Because alpha-beta generally uses less time and memory it is possible to search deeper trees and evaluate more nodes in the same amount of time. Because game-playing agents are usually stronger with deeper trees [84], an alpha-beta search will make a stronger game-playing agent given the same .

2.4.4 Iterative Deepening

Game trees often do not have a constant branching factor – meaning that each node does not have the same number of child nodes. The number of child nodes that each node has varies considerably in a normal game tree. Opening positions have large numbers of legal moves, while end game states generally have fewer possible moves. It is therefore very difficult to predict how long it will take to construct a tree of a given depth beforehand.

Because most games require a player to move within a limited, preset amount of time, it presents a problem to tree-based game playing agents. How deep should the search tree be? If the tree is too shallow, the moves made by the player will be too weak. If the tree is too deep, the time taken to generate the tree will exceed the allowed time.

A solution to the problem is a technique called iterative deepening [57]. Instead of generating a tree with a fixed depth, it is possible to start with a shallow tree and repeatedly deepen

the tree (add extra layers) until resources (for example time or memory) run out. Iterative deepening guarantees the best solution in the allowed time, even though it uses more resources (especially time and memory) than other algorithms [57].

Iterative deepening can be used to build the tree of both a minimax or alpha-beta tree.

2.4.5 Static Evaluation Function

Although the depth of the game tree is very important for the play strength of a game-playing agent, the evaluation function plays a key role in the play strength. Without an effective evaluation function it will be impossible to create a strong game agent, even with very large game trees.

Usually, evaluation functions consist of hand-coded algorithms that are encoded computer algorithms. The features that are relevant to the fitness function depend on the nature of the game that is evaluated. The complexity of the function depends on the complexity of the game.

For example, a tic-tac-toe fitness function may only evaluate the rows or columns that are still available for completing winning rows and columns. A checkers agent may take the number of checkers and kings into account, along with control of important squares and pieces on the back row. A chess fitness function is more complex: it needs to take into account the value of pieces relative to each other, as well as features that are more difficult to compute such as king safety, passed pawns, and double pawns [64].

Usually, a fitness function consists of dozens of weights and parameters that are painstakingly adjusted by hand [47][84]. The process of developing a good fitness function has been known to be a very time consuming, and often frustrating process [84]. Even so, the strongest game players in gaming history have been developed by using these techniques [47][84].

2.4.6 Limits of Tree-Based Algorithms

Tree-based algorithms perform very well for a range of two-player games. Medium-sized games that fit well in partial tree searches are popular western played games like tic-tac-toe [73], checkers [84], and chess [47]. Larger games like the eastern game of Go is at the time of this writing too large to be attempted with a tree-based algorithm [6]. For Go, even the planning algorithms discussed requires a partial tree that is simply too large to be computed by any available hardware [18].

Even though tree-based computer players can beat most human opponents for checkers [84] and chess [47], these games have not been solved yet. This means that the tree building algorithms still do not build trees that are large enough to force a winning position (or a draw in the worst case) from any arbitrary starting state. The closest candidate for solving checkers is the deep search trees of Chinook: once the search trees are deep enough to reach the end-game dictionary, that portion of the game is in effect solved [84].

Although a full game tree would enable a game agent to solve a game, the trees used in modern game agents are of limited size. A general rule of thumb is that deeper trees enable stronger game agents, but it has been shown that adding more levels to very deep trees only increases play-strength with small margins [84].

Go is an especially poor tree-based candidate due to the following: (1) Go has a huge branching factor (2), requires extremely deep searches to evaluate positions and (3) lacks an explicit evaluation function because of live and death problems [7].

Because of the limitations in existing tree-based agents mentioned in this chapter, as well as games like go and arimaa [92] that cannot be played with tree algorithms, it is useful to investigate other types of game-playing algorithms. One area of research that might promise useful to create new types of game algorithms is that of computational intelligence learning strategies.

2.5 Artificial Neural Networks

The section briefly covers the fundamentals of artificial neural network structure and training.

Artificial neural networks are based on models of biological neural systems, primarily for the learning ability of biological neural systems [43]. Similar to a biological neural system, an artificial neural network consists of artificial neurons that are linked together by weighted connections. In the case of natural neural systems, chemical neuro-transmitters convey information between neurons, whereas in artificial neural networks, numeric information is transfered by connections between neurons. Almost like their biological counterparts, the artificial neurons also fire output signals when they are stimulated by certain input signals. Even though this process is a very simplified version of what happens in a biological neural system, the results are remarkable. For the remainder of this study the terms neural network and neuron will denote an artificial neural network or artificial neuron, unless otherwise stated.

Neural networks are suitable to a large number of disciplines [72], such as classification [70][100], pattern recognition [79], pattern completion [81][85], function approximation [46][70], clustering [55], and control [37]. One reason for its success is that neural networks generalize well, i.e, correctly predict outcome under the presence of inputs not used during the training process [4].

Section 2.5.1 gives a brief overview of artificial neurons, while section 2.5.2 discusses the arrangement of neurons into neural network structures. Neural network training is discussed in section 2.5.3. Applications of neural network techniques in game agents are briefly discussed in section 2.5.4.

2.5.1 Artificial Neurons

The simplest component of a neural network is a single artificial neuron (in some cases also called a perceptron). A neuron receives a set of weighted numerical inputs. The weighted sum of the inputs is tested against a threshold to determine whether the neuron will fire or

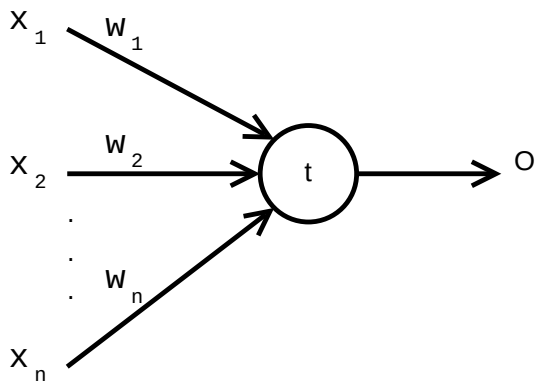

Figure 2.5: A Single Neuron

not. For complex activation functions, such as sigmoid or hyperbolic tangent, the threshold value is included in the net input signal (refer to equation(2.2)). Inputs that are not numeric, for example colors, can be converted to discrete number representations. For example, red=1, blue=2 and black=3.

As illustrated in figure 2.5 a neuron receives a net input signal described by:

$$net = \sum_{i=1}^{n} w_i x_i - t \tag{2.1}$$

where $\mathbf{x}$ is the input vector, $\mathbf{w}$ is the weight vector and t is the neuron threshold.

To simplify the net input function, augmented vectors are used [24]. The vectors for $\mathbf{x}$ and $\mathbf{w}$ are augmented to include an additional unit at index $n+1$, referred to as the bias unit. The value of x_{n+1} is always -1 while w_{n+1} denotes the value of the threshold. When using augmented vectors, equation (2.1) is redefined as:

$$net = \sum_{i=1}^{n+1} w_i x_i \tag{2.2}$$

In this way the threshold value can also be trained in the same manner that the weight vector is trained. For this study augmented vectors were used.

The output of a neuron, o, shown in figure 2.5, is determined by a mathematical function

called the activation function. The net input (given in equation (2.2)) is used as the input parameter to the activation function to determine the output of the neuron. A neuron's output is calculated as:

$$o = f(net) \tag{2.3}$$

where o is the neuron output, f is the activation function and net is the net input signal of the neuron.

Neuron Activation functions

An activation function determines the output of a neuron based on the net input signal and bias. Generally, activation functions are monotonically increasing mappings and bounded so that $f(\infty) = 1$ and $f(-\infty) = 0$. Some functions, for example the hyperbolic tangent, is bounded so that $f(-\infty) = -1$. Other activation functions, notably the linear activation function, are not bounded.

Various activation functions have been defined for artificial neurons. The most frequently used activation functions are briefly discussed below.

1. *Step function:* The step activation function, also known as the binary activation function, provides a boolean test of the net input signal against a threshold. If the value of the net input signal is greater than or equal to the threshold, the neuron outputs 1, otherwise it outputs a 0 (or in some cases -1, depending on the specific implementation). The step activation function is illustrated in figure 2.6, chart A. The step activation function is defined as (assuming augmented vectors):

$$f(net) = \begin{cases} 1 & if \quad net >= 0 \\ 0 & if \quad net < 0 \end{cases} \tag{2.4}$$

The step activation function can only produce discrete output values of 0 or 1. In cases where continuous output is required, such as function approximation problems, step

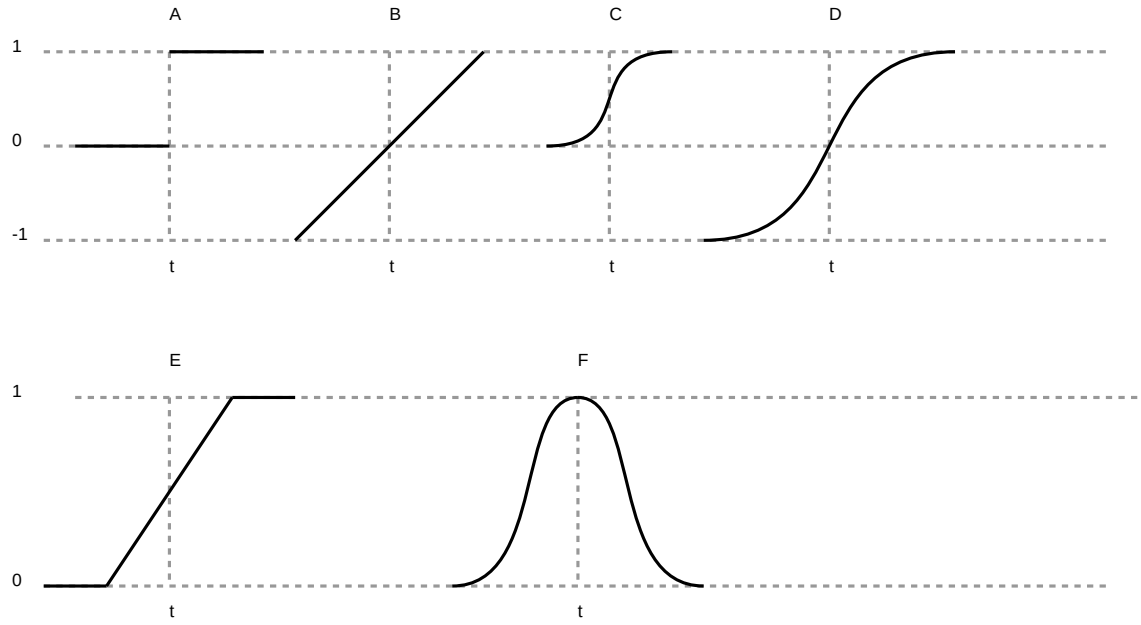

Figure 2.6: Activation Functions

activation functions are not applicable. Also, in optimization algorithms where error derivatives are used to adjust weights, the step function cannot be used.

2. *Linear function:* A linear activation function is illustrated in figure 2.6, chart B and is defined as:

$$f(net) = net \tag{2.5}$$

Linear activation functions produce output that is linear with respect to the net input signal. Optimization algorithms that rely on error derivatives can be applied to linear activation function learning. Linear activation functions are not bounded.

3. *Sigmoid function:* The sigmoid function is shown in figure 2.6, chart C. The output of the sigmoid function ranges between 0 to 1, with the gradient reaching a maximum at the threshold.

The sigmoid function is defined as:

$$f(net) = \frac{1}{1 + e^{-\lambda net}} \tag{2.6}$$

where λ controls the steepness of the function. Usually, λ is set to 1.

Because the gradient is at its steepest near the threshold, algorithms that rely on error derivates (for example, gradient descent [102]) will make larger adjustments near the threshold.

4. *Hyperbolic tangent function:* The hyperbolic tangent activation function, shown in figure 2.6 chart D, is similar in shape to the sigmoid function and is defined as

$$f(net) = \frac{e^{\lambda net} - e^{-\lambda net}}{e^{\lambda net} + e^{-\lambda net}} \tag{2.7}$$

The difference between the sigmoid and hyperbolic tangent functions is that the output of the hyperbolic tangent function ranges between -1 to 1 and that of the sigmoid function between 0 to 1.

5. *Ramp function:* A combination of the step and linear activation functions, shown in 2.6 chart E, is referred to as a ramp activation function. The ramp activation function is defined as:

$$f(net) = \begin{cases} 1 & if \quad net >= 1 \\ net & if \quad |net| < 1 \\ -1 & if \quad net < -1 \end{cases} \tag{2.8}$$

6. *Gaussian function:* The Gaussian activation function is shown in figure 2.6 chart F and is defined as

$$f(net) = e^{-net^2/\sigma^2} \tag{2.9}$$

where σ is the standard deviation.

Gaussian activation functions are generally used in radial basis function neural networks.

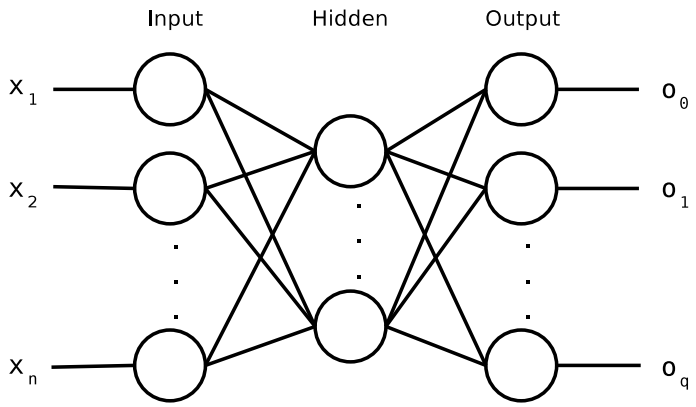

Figure 2.7: A Neuron Network

Neuron Geometry

By adjusting the input weights a single neuron can realize a linearly separable function without any error [4]. For classification problems, a neuron defines a hyperplane that separates input vectors into above-threshold values and below-threshold values. In other words, the classes will be separated between $f(net) > 0$ and $f(net) < 0$. For function approximation problems the activation function is fitted to the target function.

A single neuron, using summation net input signals, will not be able to classify linearly non-separable classification problems or approximate higher-order functions. To be able to learn more complex problems, i.e, linearly non-separable classification problems or to approximate higher-order functions, a number of neurons are combined in a neural network. Note that a single product unit (a neuron that multiplies, rather than sums inputs) can be applied to solve linearly non-separable classification problems.

2.5.2 Creating a neural network

A single neuron is not very useful on its own, but when a number of neurons are connected together to form a neural network, more complex problems can be solved. For classification problems, neural networks can learn linearly non-separable problems, whereas for function

approximation problems, neural networks can be trained to approximate higher-order non-linear functions.

A standard feedforward neural network has a layered architecture, as illustrated in figure 2.7 [43]. The layers are known as the input layer, output layer, and one or more hidden layers. Floating-point weight vectors are associated with the connections between neurons. Neural network layers are usually fully interconnected: each neuron in the hidden layer is connected to all the input neurons, while each neuron in the output layer is connected to all the neurons in the hidden layer. In some implementations, where it is known that linear dependencies between inputs and outputs exist, the input and output layers are also connected. The number of neurons in each layer depends on the nature of the problem.

The input layer of a neural network contains one neuron for each input parameter of the problem. The sole purpose is to provide an input to the neural network.

The last layer is the output layer, which is used to obtain the output vector of the neural network. The number of neurons in the output layer corresponds directly to the number of elements in the output vector of the problem space. Activation functions used in the output layer neurons are often the same as the hidden layer neurons, but may differ.

The hidden layer (or multiple layers in some instances) determines the number of boundaries a neural network can realize in classification problems or the shape of the function in function approximation problems. Activation functions used in hidden layers are usually monotonically increasing differentiable functions.

The optimal number of hidden units in a neural network is influenced by the complexity of the problem at hand, and will have a significant influence on the performance of the network, in terms of learning ability and time complexity. Often the number of hidden units is determined from experience and experimentation. However, methods have been developed to determine the optimal number of hidden units for supervised training algorithms. Examples of techniques to adjust the size of neural networks for supervised learning are pruning [82], growing [82], and regularization [33][4].

For training sets, too few hidden units may result in high training and generalization errors, due to under-fitting an statistical bias. On the other hand, too many hidden units may result in a low training error but a high generalization error due to overfitting and high variance [38][101]. For classification problems, too few hidden units may not be able to classify all the classes correctly, while to many hidden units may result in insufficient differentiation between classes.

The traditional feed forward neural network structure contains only three layers (refer to figure 2.7). Although the use of more than three layers has been investigated and is advantageous for highly non-linear problems, it has been shown that one hidden layer can solve the same problems, given enough hidden neurons in perceptron neural networks [46].

There are different neural network architectures, for example, feed forward neural networks [46], fuzzy lattice neural networks [76], product unit neural networks [19], recurrent neural networks [59], time-delay neural networks [99], self organizing maps [55] and learning vector quantizers [56].

In this study standard three-layered feed forward neural networks were used.

2.5.3 Neural Network Training

Neural networks are trained by adjusting the values of weights that connect neurons. This section covers the main types of learning, as well as techniques for adjusting neural network weights.

The main types of learning are:

1. *Supervised learning:* For supervised learning, a set of known examples called a training set, which consists of patterns of input and target output vectors, is provided. To train the neural network, the input vector of each pattern of the training set is in turn passed to the neural network. The output vector of the neural network is compared to the expected output vector of the training pattern. Weight adjustments are made

proportional to the difference between the output vector of the neural network and the expected vector in the training pattern [44].

2. *Unsupervised learning:* For unsupervised learning, the objective is to discover patterns or features in input data without a target output. A training set for unsupervised learning contains only a set of input vectors, and does not provide target output vectors. The unsupervised learning techniques are often used to cluster the training set in different categories [55][56].

3. *Reinforcement learning:* Reinforcement learning rewards neurons for good performance (or parts of a neural network) by strengthening weights connected to the neuron and punishes neurons for bad performance by weakening weights connected to the neuron.

Each pass through the training set (each training pattern is run through the neural network once) is called an epoch. Training is done over several epochs. Too few epochs may lead to a sub-optimal solution, while too many epochs may lead to overfitting or memorization of the training set (usually in conjunction with an overly complex neural network) [60]. Search spaces that contain local minima may required more epochs to converge on a global optimum. Training is terminated when stopping criteria, such as a small error, maximum number of epochs, or small changes to weights are met.

During each epoch, adjustments are made to the neural network weights. An optimization algorithm is used to determine the changes to the weight vectors. There are a range of optimization algorithms available to adjust the neural network weights. For supervised learning, optimization algorithms that rely on error derivatives, such as gradient descent [102] and leapfrog [93] can be applied. These algorithms use the difference between the neural network output and target pattern to make adjustments to the neural network weights. Supervised training algorithms that do not rely on error derivates include evolutionary computing [103][29] and swarm-based algorithms [53][83].

For unsupervised learning, algorithms are applied that do not rely on a target output

set, such as self-organizing maps [55]. Unsupervised training that makes use of coevolution [103] or particle swarm optimization in a coevolutionary training method [53][48] relies on the performance of a neural network relative to other neural networks.

This study applied a PSO coevolutionary training method to neural networks. Evolutionary and particle-swarm-based algorithms are discussed in more detail in sections 2.6 and 2.7 respectively.

2.5.4 Neural Network Game Agents

Because neural networks generalize well and have the ability to learn hidden features of a problem space [72], they can be applied to game-playing agents. Different configurations of neural networks and training strategies have been used to train game-playing agents.

The first neural network configuration takes the game board as input and produces an entire game board as output [30][65]. The game board that is produced by the neural network indicates the best move; for example the output neuron with the highest output denotes the most desirable intersection to occupy in Go [30][65].

Another approach is to apply the neural network as an evaluation function for standard game tree algorithms [10][27]. Input to the neural network is the board state, and the output is a single numerical value that represents the desirability of the given board state. The desirability is used the evaluation value for the game tree. This study uses the game tree approach (refer to chapter 3 for more detail)

The main types of neural network learning that have been applied to game agent training are:

1. *Supervised learning:* The supervised learning approach uses a set of sample games to train the neural network [95]. Usually, these games are known games of strong or master players. Unfortunately, this approach may teach the neural network techniques that are unconventional and dangerous for most players, for example, to move a queen to the center of the board in a chess opening [64][95]. Furthermore, the extreme size

of the search space of many games may not allow for a representative training set. Computational and time complexity associated with very large training sets becomes problematic with existing computing technology.

2. *Unsupervised learning:* For unsupervised learning, a neural network learns the game by playing against opponents rather than examining specific input/output patterns. The opponents may be a human players, other computer players [95], or other neural networks of the same design [10]. The play strength of a neural network, as determined by its performance against opponents, is used as the fitness value in unsupervised training approaches such as coevolution [10].

3. *Reinforcement Learning:* Temporal difference learning, a reinforcement learning technique, has been used successfully to train a backgammon player. The backgammon player managed to become a strong competitor in backgammon tournaments. [94]

The focus of this thesis is on the application of an unsupervised learning strategy to evolutionary game-playing agents.

2.6 Evolutionary Computation

The theory of natural evolution was described by Charles Darwin in [16]. The theory states that species will develop and change through a process of natural selection. Natural section describes that individuals that are better suited to environmental demands are more successful and therefore have a higher probability to survive and take part in producing offspring. Through this process, genetic traits of these individuals have a better chance to be carried over to later generations, while weak individuals (and their genetic traits) will become extinct. Surviving genetic material is refined and the species become better adapted to their environments. The process through which species change to become better adapted to their environments is called evolution. The computerized models of evolution are called evolutionary algorithms.

Evolutionary algorithms have been applied in a diverse range of problems [3], such as planning [17][49], design [39], control [17][26], classification [9][35], data mining [35], and game-agents[10][71].

Section 2.6.1 gives an introduction to evolutionary processes, while sections 2.6.2 to 2.6.7 cover different aspects of evolutionary computing in more detail.

2.6.1 Introduction to Evolutionary Computation

Simulated evolution has been applied as an optimization technique. An evolutionary algorithm makes use of a population of individuals, where each individual is a candidate solution to the optimization problem.

The features of the optimization problem are represented by parameters. The value of each parameter represents the value of a feature. The features are typically parameters to a function that needs to be optimized. The evolutionary algorithm is used to find optimal values for these parameters. The quality of an individual's parameters is referred to as the individual's fitness (or error). A function, known as the fitness function, is used to determine the fitness.

The stronger an individual is rated by the fitness function, the better the individual's chances are to be chosen for survival and reproduction through recombination. This implies that the parameters of fitter individuals have a better chance of being incorporated in future solutions. Because the process repeatedly favors fit individuals, the expectation is that the best parameters will survive and produce good solutions to the optimization problem.

Different variations of parameter representation are used. These may consist of a vector of real numbers, a binary bit string, or a tree representation, for example, vectors of real numbers may be parameters to a mathematical function, whereas tree representations are often used to represent an executable program. Vectors or binary bit strings are usually fixed-length vectors, but variable length vectors have also been applied [32][41].

A pseudo-code summary for a general evolutionary algorithm is given below:

1. Let $g = 0$ be the generation counter
2. Create a population C_g of n distinct randomly generated individuals
3. While not converged
   (a) perform recombination, if any
   (b) perform mutation, if any
   (c) select the new generation, C_{g+1}
   (d) evolve the next generation: let $g = g + 1$

There are different variations of evolutionary algorithms. In genetic algorithms, vectors of real numbers or binary bit strings are used [34], to represent genes. Variation operators, such as crossover (discussed in section 2.6.5) and mutation (refer to section 2.6.6) are applied to the vector or bit strings.

Genetic programming represents programs as trees [58]. The output of an individual is determined by traversing the tree and executing the operators it describes. Crossover takes place by copying subtrees between individuals. Mutation may add or remove subtrees to an individual, as well as altering the operators on the tree.

Evolutionary programming uses mutation as the only reproduction operator to create offspring [31]. Vectors of real numbers are used for representation of floating-on problems. However there is no restriction on representation of valuation parameters.

With reference to figure 2.8, the following sections discuss the elements of the evolutionary algorithm in more detail.

2.6.2 Population Initialization

The first step of the computational evolutionary process is to create an initial population of individuals.

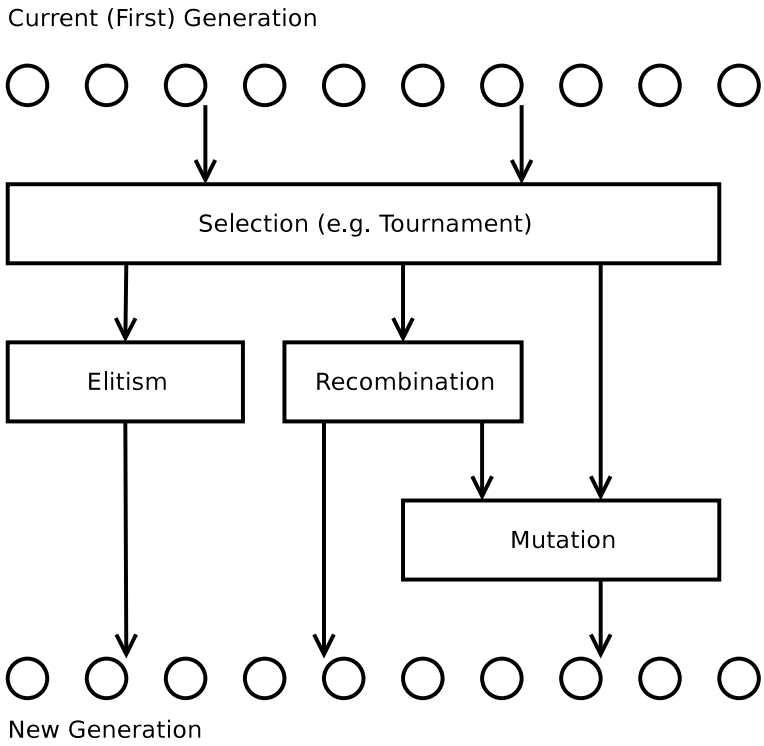

Figure 2.8: Creating a new generation for evolutionary computation

The size of the the population has an effect on the performance of the algorithm in terms of execution time and accuracy. A small population contains only a small subset of the search space and may require a large number of generations to reach an optimal solution. Although a large number of generations may be required, the time complexity per generation is lower than for larger populations. Small populations can, however, be forced to explore a larger portion of the search space by using a large mutation factor. A very large population, on the other hand, may explore a larger area of the search space in fewer generations, provided that the initial population is uniformly distributed. However, the time complexity per generation is larger.

Individuals may be randomly initialized from the domain. Uniformly distributed, random initialization ensures that the entire search space may be represented by the initial population. Prior knowledge of the search space can be used to provide a bias toward known good solutions

[63]. However, this may cause convergence on a local optimum, due to less diversity of the initial population.

2.6.3 Selection Operators

During the evolutionary process individuals are selected to survive to subsequent generations or to take part in recombination and mutation. Usually, individuals are selected based on their fitness. Various selection operators have been developed to select individuals [40][3]:

1. *Random Selection*: Random selection selects individuals randomly from the population. Selection is not based on any reference to fitness and all individuals have an equal chance of being selected. Tournament selection (see below) also makes use of random selection to select individuals to take part in a tournament.

2. *Proportional Selection*: For proportional selection the probability of an individual to be selected is proportional to its fitness. Strong individuals have a higher probability of being selected, whereas weak individuals have a lower probability of being selected. If a population contains a small number of strong individuals, these individuals can dominate the selection process by constantly being selected for crossover. The population become homogeneous too soon, with all individuals being very similar. The lower diversity may lead to premature convergence.

3. *Tournament Selection*: Tournament selection randomly selects a subset (or tournament) of individuals from the population. Based on the fitness of individuals, the best individual from the tournament is selected. The subset of individuals are returned to the pool and become eligible for selection again. The process continues until the desired number of individuals has been selected.

   Tournament selection favors stronger individuals by selecting the strongest individual from the randomly selected tournament, while maintaining a small probability that a

weaker individual can be chosen. A small number of very strong individuals have a lower probability of dominating the selection process because they are not present in every tournament, provided that a sufficiently large tournament is selected.

4. *Rank-Based Selection*: With rank-based selection an individual's probability of being selected is proportional to its fitness rank. Individuals are ordered by their fitness value and assigned a rank according to their position. Because the rank, rather than the absolute fitness, determines the probability of being selected, the probability that a highly fit individual will dominate is reduced.

2.6.4 Elitism

Elitism is the process where one or more individuals are selected from the current generation to survive unchanged to the next generation. Selection can take place using the selection operators discussed in section 2.6.3. Often the individuals with the highest fitness are selected to prevent the maximum fitness from decreasing from one generation to the next.

The number of individuals that are chosen to survive to the next generation is referred to as the generation gap. If the generation gap is 0 the new generation will consist entirely of new individuals, and it is possible that the maximum fitness of the new generation will be less than the current generation. If the generation gap is too large there will be less opportunity to increase the diversity of the search.

2.6.5 Crossover

In a population, strong genetic material, such an optimal value for a parameter, may reside in many different individuals. Strong genes from multiple strong individuals can be combined, and may produce even stronger offspring. The process in which genes from different parent individuals are combined to form new offspring is called recombination.

Recombination may be indirect, where parameters from different parent individuals are

averaged to produce offspring. The values two or more parents may be averaged to produce offspring [23][32]. In direct recombination parameters are selected from parent individuals and transfered, without change, to the offspring [2][3].

Parents (usually strong individuals) are selected using a selection criterion, such as the criteria discussed in section 2.6.3, to produce offspring. The crossover rate determines the probability that two parent individuals will produce offspring. For every parent pair a random number is generated and compared to the crossover rate. Offspring are produced only if the random number is less than or equal to the crossover rate. In some implementations, offspring are only added to the next generation if their fitness values are better than that of the parent individuals [2][3].

With reference to figure 2.9, some of the common crossover operators that are used with vector representations of individuals are listed below: [3]:

1. *Uniform Crossover*: For uniform crossover, offspring 1 randomly receives a gene from either parent 1 or parent 2. Offspring 2 receives the gene from the parent that was not selected for offspring 1. The process is repeated for every gene in the vector, so that each offspring has the same number of genes than the parents.

2. *One-Point Crossover*. For one-point crossover, a single random position in the vector of genes is selected. The genes before and after the point are swapped to produce two offspring. This approach transfers groups of genes and will therefor have a better probability of retaining behavior that are associated with multiple genes.

3. *Two-Point Crossover*. Two-Point crossover is the same as one-point crossover, but two crossover points are chosen.

The crossover operators discussed above produce two offspring.

Headless chicken crossover [50] produces offspring by pairing parents from the population with random individuals. Headless chicken crossover is not true crossover: Just like a running

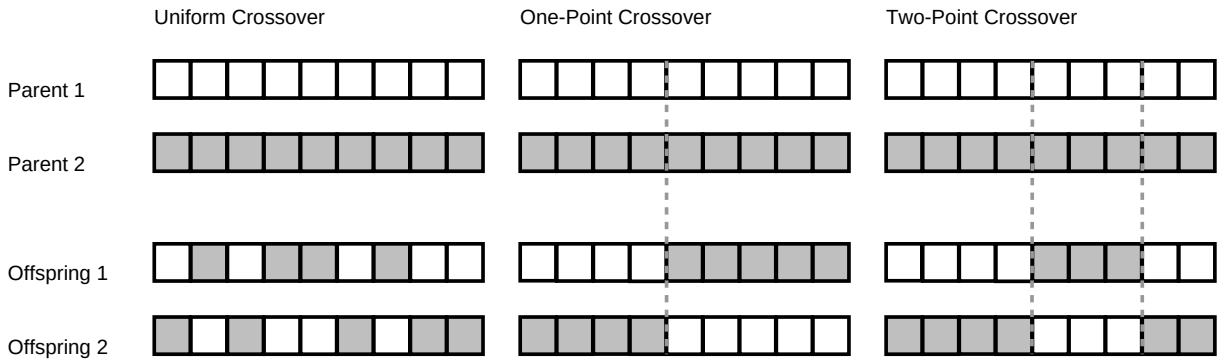

Figure 2.9: Crossover Operators

headless chicken, that shares many characteristics with a real chicken, it is not a chicken. By applying headless chicken crossover it is possible to distinguish between problem spaces that are suited to crossover and problem spaces where crossover do not add any value to genetic algorithms [50].

2.6.6 Mutation

Any given population contains a limited subset of all the available parameter values and only a portion of the search space will be examined. Mutation is used to ensure that new genetic material can be introduced to the population and that a larger portion of the search space is covered.

Mutation is applied at a specified probability referred to as the mutation rate. The mutation rate is usually a small value to prevent mutation from distorting strong individuals too much, but it is often proportional to the fitness: Weaker individuals are mutated more that strong individuals. In self adaption the mutation rate is a parameter of each individual, which may itself be mutated. The optimal mutation rate is evolved by evolutionary algorithm [3].

To mutate an individual, its genes are altered randomly: Random noise may be added to a vector of real numbers or bits in a bit string may be randomly flipped. Trees may be altered by inserting or removing nodes at random positions in the tree. In the case of

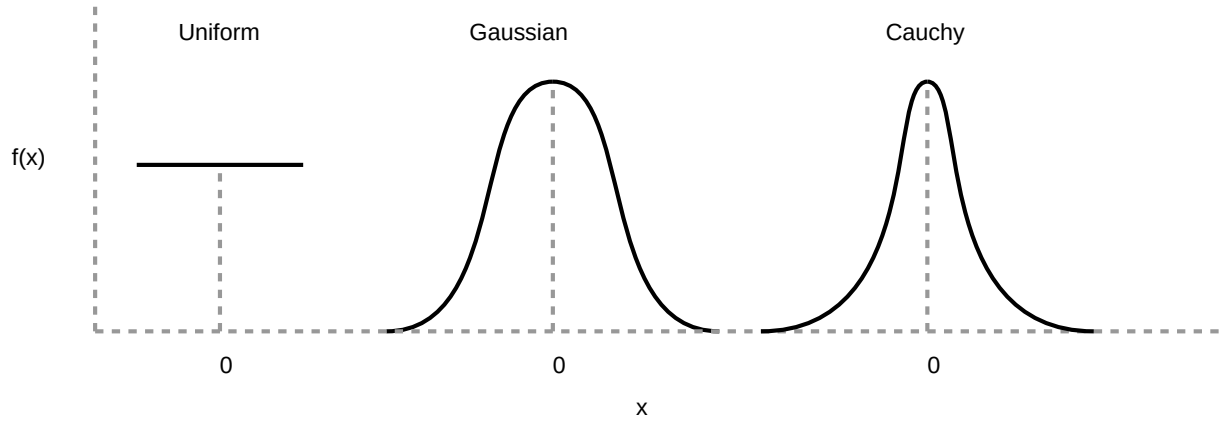

Figure 2.10: Probability Density Functions

real number representations random noise is sampled from a random distribution such as an uniform, Gaussian, or Cauchy distribution. Refer to figure 2.10 for different probability density functions.

For a continuous uniform distribution the probability distribution is such that all intervals of the same length are equally probable. The probability density function for a uniform distribution, bounded between a and b is defined as:

$$f(x) = \begin{cases} \frac{1}{b-a} & if \quad a < x < b \\ 0 & if \quad x <= a \quad or \quad x >= b \end{cases} \tag{2.10}$$

For a Gaussian distribution the probability of a value depends on the mean,μ, and the variance, σ^2. The probability density function for a Gaussian distribution is defined as:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{2.11}$$

The Cauchy distribution is a continuous probability distribution where the location parameter, x_0, specifies the location of the peak of the distribution and the scale parameter, γ, specifies the half-width at half-maximum. The probability density function for a Cauchy distribution is defined as:

$$f(x|x_0,\gamma) = \frac{1}{\pi\gamma\left[1+\left(\frac{x-x_0}{\gamma}\right)^2\right]} \tag{2.12}$$

In this study, both uniform and Gaussian distributions were used.

2.6.7 Coevolution

In a standard evolutionary algorithm, the fitness of an individual is determined by a predetermined absolute fitness function. The absolute fitness function defines how close an individual is to an optimal solution in the search space. In optimization problems such as game-playing agents [10][12] or military strategies [24], it is difficult to define an absolute fitness function.

Concepts of coevolution[20] can be applied to problems in which an absolute fitness is not available. In coevolution, the fitness of an individual is measured as its relative performance against other individuals. The relative performance model can be competitive (for example, predator and prey relationships in nature) [45] or cooperative (for example, symbiosis in nature) [77]. For this study a competitive coevolutionary model has been used.

To illustrate competitive coevolution, examples of interactions between natural enemies have been used [45]. Consider a plant that evolves a tougher exterior to protect itself from plant-eating insects. In following generations the insects evolve stronger jaws to chew through the tough exterior. The plants may evolve a poison to kill insects with strong jaws. New generations of insects can evolve an enzyme to digest the poison. The process becomes a perpetual arms race between two rival populations.

There are different models to define the population dynamics of a coevolutionary population. For competitive coevolution a global model [8] consists of a single population where all individuals compete against each other.

For cooperative coevolution an island or neighborhood model is used. In an island [42] model, different sub-populations of individuals evolve in isolation. Each island evolves as a global model with competition between native individuals. In addition, individuals are allowed

to migrate between islands to increase diversity. Because of the parallel nature of the island model, remote computers connected through a communications framework can be applied to run the optimization in a distributed framework.

Similar to the island model, a neighborhood model [69] has a number of different sub-populations. The main difference is that an individual belongs to more than one neighborhood. Overlapping neighborhoods can increase the speed of convergence, but reduces opportunities for parallel processing.

A population model, discussed above, defines a pool from which opponents will be drawn, such as an island or neighborhood. The process of selecting competitors from the pool in order to quantify the relative fitness of an individual is referred to as fitness sampling. Different fitness sampling methods have been defined:

1. *All-versus-all sampling*: All-versus-all sampling matches each individual with all the opponents in the competition pool. Although all-versus-all sampling provides the most accurate assessment of an individual's relative fitness, the performance penalty is significant, especially with large populations. Time complexity increases with $O(n^2)$ as population size is increased.

2. *All-versus-best sampling*: All-versus-best sampling only matches an individual with the best opponents. The time complexity increases linearly with larger population sizes. Linear time complexity allows for greater diversity by using larger populations, but the benefits may be negated by a few strong opponents that reduce diversity by dominating the coevolutionary process.

3. *Random sampling*: Random sampling matches individuals with a fixed number of randomly selected opponents. Although random sampling provides a less accurate assessment of the relative fitness of individuals than all versus all sampling, time complexity only increase linearly with increases in population size.

4. *Tournament sampling*: Similar to tournament selection discussed in section 2.6.3, tournament sampling uses a tournament of randomly selected individuals to select strong opponents.

5. *Shared sampling*: In shared sampling, a sample of opponent individuals are selected with maximum competitive shared fitness. See competitive fitness sharing below for a description of competitive shared fitness.

Fitness sampling provides an individual with opponents to evaluate its relative fitness [80]. Relative fitness can be calculated in the following ways [80]:

1. *Simple fitness.* If simple fitness is used, the relative fitness is defined as the sum of successful outcomes against all opponents. In other words, an individual gets one fitness point every time it beats an opponent.

2. *Fitness sharing.* Fitness sharing extends the simple fitness model by dividing the simple fitness value by the number of structurally similar opponents. Similarity can be defined as the number of individuals that also beat similar individuals. The fitness sharing model favors individuals that are unusual or more diverse, i.e, individuals that are dissimilar from the rest of the population.

3. *Competitive fitness sharing.* Competitive fitness sharing aims to reward individuals that manage to beat opponents that are beaten by very few individuals. An individual's simple fitness is divided by a metric that measures the individual's similarity to other individuals. For example, an individual's fitness is inversely calculated to the number of fellow individuals that could beat a particular opponent, thereby rewarding more points to an individual that was able to beat an opponent no, or few other population members could beat.

The main benefit of an coevolutionary process, i.e, the use of a relative fitness instead of an absolute fitness function, is also its biggest drawback: The process can be skewed by a "one-

hit wonder", known as the Buster Douglas effect (in reference to a world heavyweight boxing champion who lasted only nine months) [5]. In the absence of a universal fitness function a single individual can outperform the majority of competitors by exploiting general deficiencies of the population. A thorough benchmark or exposure to other coevolutionary environments will expose the lack of robustness of a Buster Douglas individual.

2.6.8 Training Neural Networks Using Evolutionary Algorithms

Evolutionary algorithms have been used successfully to train neural networks [103][29]. This is also true for training neural networks that are applied in game-playing algorithms [10][71].

For neural network training using evolutionary algorithms, each individual in the population represents a candidate neural network. The genetic makeup of each individual consists of a vector that contains all the weights of the neural network. The vector of weights can be encoded as a vector of bits or floating-point numbers that represent the genes of the neural network. The optimization problem to be solved is to find the best values for the neural network weights, so that a fitness function is optimized.

The initial population contains a set of randomly initialized neural networks. In other words, each individual in the initial population consists of neural networks with randomly selected weights.

The fitness of each neural network is evaluated by the output that it produces. For supervised learning the error of the actual output compared to the target output is used, while the relative fitness is used for unsupervised learning. Individual neural networks with a smaller error or higher rank, have a higher probability to take part in producing future generations. The standard evolutionary reproduction and mutation operators are applied to the vector of neural networks weights, depending on the evolutionary algorithm that is used. In the final generation, the individual with the strongest performance is selected as the best solution for the neural network weights.

For coevolutionary training, each individual represents a candidate neural network. The

fitness of each neural network is determined by its performance against other neural networks. In each generation, neural networks are sampled from the population by applying one of the sampling methods discussed in section 2.6.7.

The sample method matches neural networks to compete against each other to determine the relative fitness of each neural network in the population. The output of a sampled neural network is compared to the output of an opposing neural network, for example, playing a game [10][71]. Opponents are matched against each neural network, and the relative fitness calculated as discussed in section 2.6.7.

Candidate neural networks with a higher relative fitness have a better chance of survival and producing offspring. In the final generation, the individual with the highest relative fitness is returned as the best candidate neural network.

2.7 Particle Swarm Optimization (PSO)

PSO is a population-based, stochastic, optimization technique based on observations about natural swarms. The basic idea of PSO can be illustrated by visualizing a flock of birds in flight, where the direction of each bird is influenced by its own position and the position of neighboring birds (or particles) [22][53].

PSO has a wide range of applications such as microbiology [22] and simultaneous optimization of both discrete and continuous variables [36]. An often-cited application of PSO is neural network optimization [96]. PSO neural network training has been used, amongst others, to solve the XOR problem [51], to classify if human tremors are the result of Parkinson's Disease or essential tremor [21], and to train game agents [71].

Section 2.7.1 gives a brief introduction to PSO, while section 2.7.2 discusses PSO topologies. Section 2.7.3 gives a brief overview of PSO vector adjustments. Different PSO algorithms are discussed in section 2.7.4.

2.7.1 Basics of Particle Swarm Optimization

PSO uses a swarm of particles, where each particle is a candidate solution to the optimization problem under consideration. Each element of a particle's position vector, $\mathbf{x}$, represents one parameter of the problem that is being optimized.

Adjustments to each particle's position, and by implication the parameters of the candidate solution, is determined by the particle's velocity vector. To move a particle to a new position, the velocity vector is added to a particle's current position.

Vector adjustments depends on a particle's own position, as well as the position of other particles in the swarm. The swarm topology determines which particles have a direct influence on a given particle's direction.

2.7.2 Neighborhood topologies

Social interaction in a PSO swarm exchanges information about the search space among particles. The social component of each particle's velocity vector moves the particle toward a social best solution.

Different PSO social topologies have been developed (refer to to figure 2.11) [52]:

1. *Star topology.* In a star topology, each particle communicates with all other particles. All particles will move toward the best particle in the swarm.

2. *Ring topology.* Each particle only communicates with its immediate neighbors and moves closer to the best particle in that neighborhood. Because neighborhoods overlap the entire swarm converges to a single point.

3. *Wheel topology.* In a wheel topology a single particle, called the focal particle, is connected to all other particles. This topology isolates particles from each other, so that all information is exchanged via the focal particle.

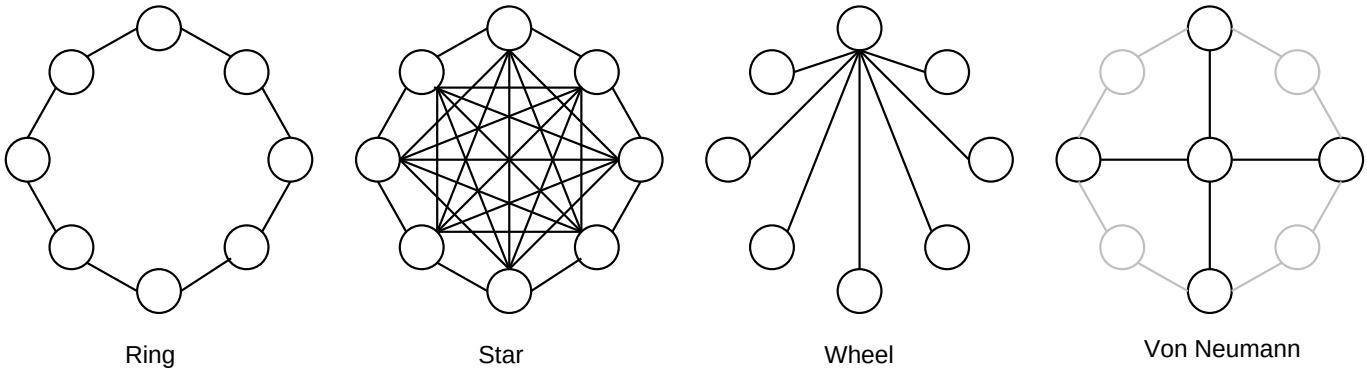

Figure 2.11: Neighborhood topologies for PSO

4. *Von Neumann topology* [54]. The von Neumann topology extends the one-dimensional lattice of a circle with a two-dimensional lattice. Each particle is connected to its immediate left- and right-hand neighbors, as well as neighbors above and below it in variable space.

2.7.3 PSO Vector Adjustments

In each iteration the position vector of each particle is adjusted by adding a step size to the position vector. Position vectors are updated using:

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1) \tag{2.13}$$

where $\mathbf{x}_i(t)$ is the position vector for particle i at time-step t; $\mathbf{v}_i(t+1)$ is the velocity for particle i. At time-step $t = 0$, $\mathbf{v}_i(t)$ is usually 0 or may be initialized to a small random vector.

The adjustment of a particle's velocity,$\mathbf{v}_i(t+1)$, is determined by the following components:

1. *Cognitive component.* The cognitive component represents the knowledge that a particle gathers about the search space while it moves through the search space and examines

different solutions. The best solution that a particle has discovered thus far is stored as the particle's personal best solution.

2. *Social component.* Particles exchange social knowledge with each other about the best solution discovered by other particles in the swarm. The social component will move a particle toward the best solution discovered by the swarm (or a subset of the swarm, depending on the neighborhood topology).

3. *Inertia component* [89]. The inertia component adds momentum to a particle's velocity by adding a fraction of the previous velocity. Larger inertia values prevent the particle from changing direction easily (exploiting existing knowledge), while smaller inertia values allow the particle to change direction often (exploring a wider area).

The velocity is the vector that determines the direction and magnitude of the movement of particle i, calculated as:

$$\mathbf{v}_i(t+1) = w\mathbf{v}_i(t) + c_1 r_{1i}(t)[\mathbf{y}_i(t) - \mathbf{x}_i(t)] + c_2 r_{2i(t)}[\hat{\mathbf{y}}_i(t) - \mathbf{x}_i(t)] \tag{2.14}$$

where $\mathbf{y}_i(t)$ is the position vector, at timestep t, of the personal best position for particle i; w donates the inertia weight; c_1 and c_2 are referred to as the acceleration coefficients; $\hat{\mathbf{y}}_i(t)$ is the best social solution of particle i.

The acceleration coefficients are used to determine step size toward the personal and neighborhood best solutions of particle i. Finally, r_1 and r_2 are random vectors sampled from a $U(0,1)$ distribution.

Correct values for w, c_1, and c_2 are important to ensure convergence of the PSO algorithm. In combination, these parameters determine the convergence behavior of PSO algorithms [14]. It has been shown that values where $w > 0.5(c_1+c_2)-1$ and $w < 1$ will guarantee convergence. Values where $w <= 0.5(c_1 + c_2) - 1$ are not guaranteed to converge [96].

2.7.4 PSO Algorithms

A pseudo algorithm for PSO is given below:

1. Initialize the swarm, P_l, with random particles for timestep $t = 0$. Each vector $\mathbf{x}_i(t)$ is the position vector of particle i at timestep t.
2. For each particle, set the personal best to the current position.
3. For each particle, set the velocity to 0.
4. While not converged
   (a) Evaluate the performance of each particle, using its current position $\mathbf{x}_i(t)$.
   (b) Compare the performance of each particle to its personal best, and update the personal best if the current position is stronger.
   (c) Compare the performance of each particle to the best particle in the social neighborhood, and update if necessary (refer to section 2.7.2).
   (d) Update the velocity vector for each particle (Equation (2.14)).
   (e) Move the particle to its new position (Equation (2.13)).

Different approaches have been developed to compute the velocity vector for a PSO algorithm. Commonly used algorithms are the Global Best (*gbest*) PSO, the Local Best(*lbest*) PSO, as well as the Guaranteed Convergence PSO (GCPSO) [22][96]. The algorithms are described below.

Global Best

The *gbest* PSO algorithm uses a star topology. All particles move toward the best position that has been discovered by the swarm, given as $\hat{\mathbf{y}}_i$ in equation (2.14). The best position is referred to as the global best position. Adjustments to a particle's position are a function of the global best solution as well as its personal best solution, given as $\mathbf{y}_i$ in equation (2.14).

If the global best position stays constant (no particle evaluates to a better solution), the algorithm will stagnate [14][96]. If the best position changes often, the swarm flies through the search space until a good solution is found.

Local Best

The *lbest* algorithm follows a circle or ring topology with overlapping neighborhoods. With reference to equation (2.14), the local (or neighborhood) best is denoted by $\hat{\mathbf{y}}_i$. The neighborhood size may vary from 2 to the swarm size. If the neighborhood size is equal to the swarm size the *lbest* algorithm is equivalent to the *gbest* algorithm. Because neighborhoods overlap, the entire swarm eventually converges.

Lbest has the advantage that a larger area of the search space is traversed. Because a larger area of the search space is covered the algorithm may take longer to converge but may find a more optimal solution when compared to *gbest* [88].

GCPSO

Gbest and *lbest* PSO implementations, in some cases, may lead to premature convergence or convergence on a sub-optimal solution [96]. If a particle's current position is equal to both the neighborhood (or global) best and personal best solutions the following equation holds:

$$\mathbf{x}(t) = \mathbf{y}(t) = \hat{\mathbf{y}}(t) \tag{2.15}$$

With reference to equation (2.14) the velocity of a particle will only be influenced by the inertia weight component. This will cause the particle to stagnate and the algorithm converges prematurely.

This problem is solved by the guaranteed convergence PSO (GCPSO), where the behavior of the standard PSO algorithm is altered at the global best solution. This forces the global best particle to do a local search and not to stagnate when $v \rightarrow 0$.

In GCPSO, the global best particle is updated using equation (2.16) when equation (2.15) is true. All other particles are updated using the standard updates given in equations (2.13) and (2.14).

$$\mathbf{x}(t+1) = \hat{\mathbf{y}}(t) + w\mathbf{v}(t) + p(t)(1 - 2r_2(t)) \tag{2.16}$$

The term $p(t)(1 - 2r_2(t))$, forces the particle to perform a search in the immediate vicinity of $\hat{\mathbf{y}}$, in the case of equation (2.15). $\mathbf{v}(t)$ is the particle velocity, given in equation(2.14). The algorithm searches in the immediate area of $\hat{\mathbf{y}}(t)$, while the diameter of the search is controlled by $p(t)(1 - 2r_2(t))$. An initial value of $p(t) = 1.0$ has been shown to produce good results. $p(t)$ is adjusted after each time step, according to whether a good solution has been found or not [96].

2.7.5 PSO Training of Neural Networks

PSO algorithms have been applied to training neural networks [53][83].

For neural network training each PSO particle represents a single candidate neural network. The position vector of each particle is the vector of all the weights of a neural network. The swarm is initialized with a set neural networks with random weights.

Each neural network is evaluated by the output that it produces. For supervised learning the error of the actual output, compared to the target output is used, for example the mean squared error that the candidate neural network produces for a training set. The error is used to determine if a particle's candidate solution should replace the personal or neighborhood best solutions.

For unsupervised training the performance of a particle's candidate solution relative to other solutions may be used to determine a particle's strength. Relative performance may be calculated using coevolutionary principles, discussed in section 2.6.7. Candidate solutions with a better relative strength are used for personal and neighborhood best solutions.

The standard PSO equation is used to update particle positions. Neural networks that

produce better or more desirable output will be used as global best, local best or personal best solutions.

2.8 Game Agents

This thesis considers applications of computational intelligence techniques to the games of tic-tac-toe and checkers. For this purpose, existing tic-tac-toe and checkers game agents are overviewed.

2.8.1 Tic-Tac-Toe

Tic-Tac-Toe (noughts and crosses) is a very simple game compared to other board games. Dozens of implementations exist to play tic-tac-toe – some of them show novel approaches to game-playing agents [1], while others use simple brute-force mechanisms.

One of the most popular implementations is a minimax search tree, which builds a tic-tac-toe game tree to find the best move. Because the search space for tic-tac-toe is small, the entire game tree can be constructed, which allows the game agent to make perfect moves, that is,a moves that would force a win at best or a draw at worst. The entire game tree can be constructed in a small amount of time, and the game can easily be solved. Through examination of the game tree a simple dictionary for perfect play can be constructed.

Another approach to creating a tic-tac-toe game agent is to build a rule-based system, for example, to use the set of rules presented in section 2.2. Game players based on these rules consist of a simple stimulus-response agent.

Tic-Tac-Toe also gets the attention of computation intelligence game agents, such as co-evolutionary algorithms and coevolutionary approaches[11].

2.8.2 Chinook

Chinook is currently the strongest checkers player, including humans and other computer programs [84]. Chinook managed to pose very strong opposition to Marion Tinsley, universally considered to be the greatest human checkers player of all time, before Tinsley was forced to resign due to health reasons [84].

Chinook primarily relies on a state-of-the-art parallel alpha-beta search engine, allowing it to utilize multiple processors to build different branches of the search tree simultaneously. This alpha-beta search engine can search to an average ply depth of 19.

Certain game states are more important than others, for example, where a specific move can force a win for either player. It is important for a game-playing agent to identify and respond a turning point in the game. In Chinook, large tactical lookup tables are used to identify critical points, where extra tree expansion is necessary. These tactical tables consist of about eight million abstracted positions that represent important game states that warrant deeper searches. The Chinook tree is therefore expanded to deeper levels on more important game states.

To evaluate the positions of the game tree, Chinook utilizes an advanced hand-coded evaluation function. The function uses more than 20 features of the game state that are weighted to convey their relative importance. Attempts were made to automate the optimization of the different weights, but in the end, the only technique that proved consistent was hand tuning of the heuristic weights [84].

Chinook utilizes a large opening database for game openings. The database was initially created from known openings in common checkers literature, but these openings contained some erroneous moves due to errors in the literature as well as mistakes in human interpretations. To improve the opening database, the data were evaluated with very deep search trees. Evaluating the openings improved the opening database in terms of adding new unknown variations, as well as removing human errors and inconsistencies.

The last, but perhaps most important, of Chinook's strengths is an end-game database that

contains solutions to all games with 10 pieces or less on the board – a compressed database of 6 Gigabytes of data. This means that if Chinook can reach the end-game it effectively plays solved games from there on. It occasionally happens with the deep alpha-beta search that Chinook can look ahead far enough to reach the end-game database after only a small number of moves have been made.

Chinook has shown that an essential brute-force approach can be applied to create the strongest checkers player in the world and it is probably close to solving checkers. Although Chinook's checkers performance is a remarkable achievement, it is relatively uninteresting in terms of computational intelligence. Chinook has been developed as a very efficient checkers move evaluation algorithm, but it cannot gain experience or adapt to its environment. Chinook cannot learn playing strategies. A very different approach is needed to develop an adaptive, intelligent agent.

2.8.3 Blondie24

Blondie24 was developed in the late 1990's as an experimental project to prove that evolutionary algorithms can learn from their own experience to play board games at a level that is comparable with human experts [10][27]. The aim was to produce a checkers player that learned to play checkers without any prior expert knowledge, other than the rules of the game. Human intervention was kept to a minimum.

Blondie24 consists of three main components: An alpha-beta search tree for the checkers game tree, a static evaluation function based on a neural network, and a coevolutionary algorithm for training the neural network. Refer to figure 2.12 for a representation of the different components.

For training purposes, the alpha-beta search tree is a standard checkers search tree to a ply depth of 4. In game play, the alpha-beta search tree used an iterative deepening algorithm that reached a 4 ply depth, and for narrow trees it could reach 6 ply, or occasionally more.

The neural network is used as the static evaluation function of the leaf nodes of the alpha-

beta search tree. The input layer of the neural network receives the 32 board positions of a checkers board as inputs, along with the difference in the number of pieces left for each player. Inputs of checkers is set to 1 while the value of a king is evolved with the coevolutionary process. A single neuron is used in the output layer to give the desirability of any given board state. This desirability value is used as the static evaluation for leaf nodes in the alpha-beta search tree.

Hyperbolic tangent ($tanh$) functions are used as activation functions for both the hidden and output layers. Therefore, the output of the neural network is in the range $(-1, 1)$. Values closer to 1 denote a favorable or strong board position for player 1 and values closer to -1 denote a favorable position for the opponent.

The training process consists of a coevolutionary approach to neural network training. A population of randomly initialized neural networks is used. Each neural network is matched in a game of checkers against five randomly picked opponents and awarded points based on its performance (1 for a win, -2 for a loss, and nothing for a draw). The game is played by building an alpha-beta search tree for each opponent. The first neural network is used as the static evaluation function for the first alpha-beta search tree, and the second neural network for the second alpha-beta search tree. The neural networks determine the moves of the players without being altered for the duration of the game.

The accumulated score of each neural network is then used to determine the play strength of the neural network. Only after all the neural networks have been evaluated are weights altered to form a new generation of neural networks. To create the new generation, the weakest half of individuals are removed from the population. The strongest half are retained, and each surviving individual generates one offspring, bringing the total number of individuals back to a full generation. Offspring are created by using an evolutionary programming process, and are mutated with Gaussian random noise. The training process lasted for roughly six months on a Pentium II (450 MHz computer).

It should be noted that, although the play strength was tested against humans, Blondie24

relied exclusively on this training process to learn checkers. No human interference or knowledge was involved to determine a move. Even though Blondie24 did not have a human trainer, and in fact its creators were poor checkers players, it managed a remarkable play strength.

To improve the play strength even further, an additional hidden layer was added to the neural network. The additional hidden layer is only partly interconnected to provide some of the two-dimensional features of a checkers board to the neural network. Because the neural network gained knowledge about positions of pieces relative to each other its performance increased significantly. The final version of Blondie24 was rated at the "expert" level, is able to beat most human opponents, and even the odd master player [27].

The effectiveness of Blondie24 was tested by playing games against human opponents over the Internet. This approach is unfortunately very time consuming, for a human mediator has to manage the contact on behalf of the computer player. Furthermore, it is also relatively subjective because the play strength of the agent will be measured against the ratings of human players, who are by nature fallible and influenced by external factors.

With Blondie24, it has been shown that neural-network-based players can achieve remarkable play strength, even when trained from zero tactical knowledge and no interference from humans. The coevolutionary approach turned out to be very effective [27].

2.9 Summary

There exist various techniques for creating game-playing agents. Some of the popular and successful methods were reviewed in this chapter, for example, exhaustive search, tree-based heuristics, training on examples, and coevolutionary training. Although these techniques have been developed and refined for almost as long as the existence of computers, complete solutions for most games still elude researchers.

There were major breakthroughs in the form of tree-based algorithms. Chinook managed to become the human-computer checkers world champion [84] and Deep Blue beat Garry

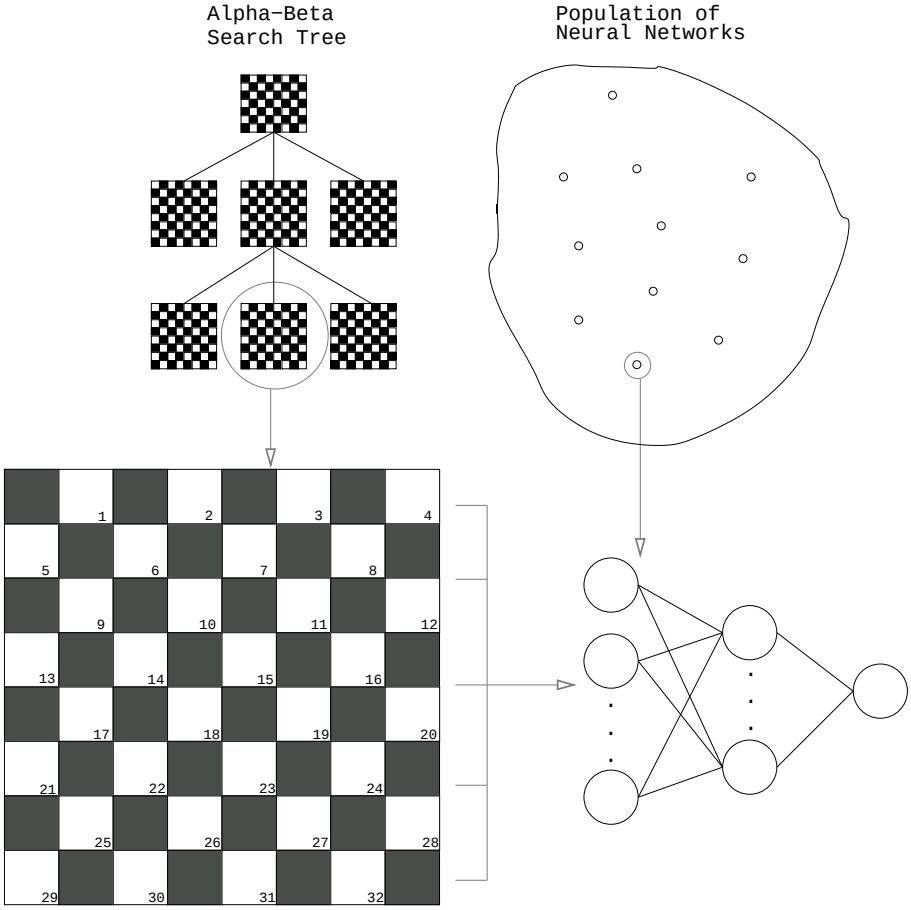

Figure 2.12: Blondie24 Architecture

Kasparov in a 1997 match [47]. Deep Junior, in contrast to Deep Blue, did not run on specialized hardware and managed to force Garry Kasparov to a (not often cited) three-all draw [13]. However, even with the monumental growth in computer speed and storage size, tree-based algorithms have probably met their match in games such as the Chinese game of Go [7] and arimaa [92].

To develop algorithms that are more generally applicable and more scalable than tree-based algorithms, researchers have looked at learning algorithms. Neural networks seem to be a popular choice because of their relatively simple implementation and large potential because of their learning and generalization abilities.

Early neural network attempts showed a slow and somewhat disappointing start, but with the introduction of Blondie24, neural networks trained with coevolution were suddenly placed on the map of game-playing agents. Blondie24 proved that it is possible for a computational intelligence agent to learn to play a game without expert knowledge introduced by humans.

For this study it was decided to build on the foundation that was created in Blondie24. The coevolutionary approach of Blondie24 inspired the idea that PSO algorithms may also be able to train game playing agents. This study explores the learning abilities of PSO algorithms for game-playing agents.

In the next chapter a general framework for coevolutionary neural network game playing agents is introduced. This framework can be applied to a wide range of games, utilizing almost any neural network training process. The framework is then applied to tic-tac-toe and checkers.

Chapter 3

Co-Evolutionary Training of Game Agents

This chapter presents a general framework of a coevolutionary approach to train neural networks as static evaluation functions for game tree algorithms.

3.1 Introduction

Training of a game-playing agent requires the agent to learn strategies and tactics of the game by itself. The process of learning a game strategy for this study does not rely on either human knowledge or existing databases to compute moves, but must rather rely on experience gained by repeatedly playing the game. Section 3.2 summarizes the components used in this study to train a game-playing agent. It is shown how both an evolutionary algorithm and a PSO algorithm can be used within the same framework to train neural networks as evaluation functions. Section 3.3 summarizes the implementation differences between the general framework and Blondie24.

3.2 Components

The framework used for training game-playing agents in this study is similar to that of Blondie24 (refer to section 2.8.3). This framework consists of the three main components: a game tree, a static evaluation function implemented as a neural network, and a coevolutionary training algorithm. The relationship between these components is illustrated in figure 2.12.

In these general terms, the framework can be applied to an arbitrary two-player, perfect knowledge, board game, by simply using a game tree and neural network input appropriate for that game. Either minimax or the alpha-beta search can implemented, with the game tree to a desired ply-depth.

This study assumes unsupervised training that relies on gaining knowledge from competitive game play rather than focusing on existing example games. In other words, the training relies on experience and not on teaching. Different training algorithms may be applied to unsupervised neural network training, for example, evolutionary programming, similar to the process used applied in Blondie24 [27], or the PSO training discussed in chapters 4 and 5.

3.2.1 Training Using Evolutionary Programming

While any evolutionary algorithm can be appropriate to adjust the neural network weights, this study concentrates on the coevolutionary programming approach. Each individual represents a candidate neural network for a game tree static evaluation function.

To determine the fitness of an individual, a round of games is played where the neural network is used as a static evaluation functions. A round consists of playing each individual against randomly picked opponents (this study uses a tournament size of five). After each game the winner receives a score of 1 and the loser a score of -2. A larger proportional score for losing a game punishes losers similar to the Blondie24 implementation [27]. Nothing is awarded for a draw. The scores are accumulated for all the games of a round.

After the round of games is completed, the accumulated scores are used to determine the best individuals. High accumulated scores are associated with strong individuals, while low scores denote weak individuals. Strong individuals are allowed to create offspring, while the weak individuals become extinct. After training, the best individual is chosen as the final game playing agent.

While many schemes exist for creating a new generation, this study follows a similar scheme than the one used in Blondie24: The lower (or weakest) half of the individuals is discarded. The upper (or strongest) half is allowed to carry over to the next round. To replace the weak individuals, each of the strong individuals creates one offspring. The offspring is a duplicate of the parent individual, but randomly mutated by adding random noise (in this study both Gaussian and uniform noise have been used) to the neural network weights.

The evolutionary programming approach is summarized as follows:

1. Initialize the population of s random individuals
2. For each iteration
   (a) Clear the scores of all individuals
   (b) For each individual $\mathbf{x}_i$
      i. Pick k random opponents from the rest of the population
      ii. For each opponent $\mathbf{o}_j$
         A. Let $\mathbf{x}_i$ be player 1
         B. Let $\mathbf{o}_j$ be player 2
         C. Play the opponents against each other using the game tree, with the respective neural networks as evaluation functions.
         D. Add 1 to the score of the winner and subtract 2 off the score of the loser; do nothing for a draw
   (c) Remove the weakest $s/2$ individuals

(d) For each of the remaining $s/2$ individuals:

i. Create a duplicate $\mathbf{x}'_i$

ii. Randomly adjust the weights $\mathbf{x}'_i$

iii. Add $\mathbf{x}'_i$ to the new population

3.2.2 Training Using PSO

This thesis proposes that PSO be used to adjust the neural network weights. In this case a swarm of particles, where each particle consists of a neural network, is flown through the hyper-dimensional search space. Neural network weights are adjusted using the velocity and position update equation as reviewed in section 2.7.

Finding the strongest agent (particle) differs from the evolutionary programming approach. A competition pool is created that consists of all current particles (neural networks) and their recorded personal best positions. For the PSO approach, not only do the particles compete, but also the personal best solutions for each particle.

Because the current position of a particle changes with every time step, personal best solutions should be evaluated against the new particle positions. For example in step 1 a candidate (personal best) solution may achieve a relative score of 5, but in step 10 the same candidate solution may only score 2. Adding personal best solutions to the competition pool compensates for the fact that opponents get stronger.

Each neural network in the competition pool competes against other neural networks. For this study each neural network competes against five neural networks from the competition pool. Each opponent is randomly selected. After each individual in the competition pool had an opportunity to compete, the accumulated score is used to find the best neural network among the current particles and personal best particles. If the score of a particle's current position is higher than the score of its personal best solution, the personal best solution is replaced with the current position.

Because both particles and their personal best solutions compete in the competition pool, the number of games played in each round is roughly double that of the evolutionary programming approach. When comparisons are drawn between the play strength of different approaches the number of games played have to be taken into account. The measuring criteria in following chapters have been adjusted to compensate for this inequality.

The PSO approach is summarized as follows:

1. Initialize a swarm of s random individuals
2. For each iteration
   (a) Clear the scores of all particles and best solutions
   (b) Create a competition pool, consisting of all particles and their personal best positions
   (c) For each solution $\mathbf{x}_i$ in the competition pool
      i. Pick k random opponents from the competition pool
      ii. For each opponent $\mathbf{o}_j$
         A. Let $\mathbf{x}_i$ be player 1
         B. Let $\mathbf{o}_j$ be player 2
         C. Play the opponents against each other.
         D. Add 1 to the score of the winner and subtract 2 off the score of the looser; do nothing for a draw
   (d) Compute the strongest individual $\hat{\mathbf{y}}$ in the neighborhood
   (e) For each particle i in the swarm:
      i. Update its personal best position, and call it $\mathbf{y}_i$
      ii. Adjust $\mathbf{x}_i$, using equation (2.13).

In this study different variations of PSO have been applied to game playing agents, namely *gbest* PSO, *lbest* PSO and *lbest* GCPSO.

3.3 Differences to Blondie24

There are differences between the approach of Blondie24 and the implementations used in this study. This section discusses these differences.

Neural Network Input Layer

Material advantage (i.e, the difference between the number of pieces on the board for each player) is very important in checkers. A good material advantage usually leads to a win. In games like tic-tac-toe and go, players take turns to place tokens, and therefore always have the same number of tokens on the board. A material advantage therefore never exists for tic-tac-toe and go and it is not meaningful to use material advantage in the evaluation function [6]. Blondie24 used the material advantage as an extra parameter. Even though material advantage is important in some games, it has been dropped from this study to make the techniques more generally applicable.

In games such as go the tokens or pieces on the board are all similar, but in other games (especially games such as chess and Arimaa) the values of the pieces differ. Blondie24 uses an extra input parameter to train the different values for different pieces. In this study the values for pieces are not trained but given as discrete values: The input value for a checker is always 1 and 2 for a king. Neural networks are able to learn the relative values of these discrete values.

Because board states may look very similar, but in actual fact favor one player or the other, the neural networks used in this study use the player who has to make the next move as an additional input parameter.

Neural Network Hidden Layers

For the Blondie24 agent ten hidden units were used [27]. For the neural networks in this study, different experiments were done, each with a different number of hidden units. From these

experiments good choices for the number of hidden neurons can be determined.

For the final version of Blondie24, an extra hidden layer was added to the neural network. The connections for this layer were created in such a way that they added some knowledge of the two-dimensional aspects of a checkers board to the checkers player. To maintain the general applicability (i.e, the ability to apply these techniques to any game) of techniques introduced in this study, no attempt is made to add this knowledge to the neural networks.

Neural Network Output Layers

Blondie24 uses hyperbolic tangent functions as activation functions in both the hidden and output layers. The output of the neural network is therefore in the range $(-1, 1)$. Values closer to 1 indicate a good position for one player, while values closer to -1 indicate a good position to the other player. This study uses sigmoid activation function with output in the range $(0, 1)$. Values close to 1 indicate a good position for the player specified in the input layer, while values close to 0 indicate an undesirable position to the player indicated in the input layer.

Search Tree

Blondie24 uses a game tree of ply four, but the implementations used in this study only search to one ply. The ply-one search depth tests the worst-case scenario for a game learning agent and encourages players to attempt to discover aspects of the game and not to rely on the information provider by a large number of leaf nodes.

A smaller tree also reduces training times so that more experiments can be carried out in a reasonable amount of time.

Training Algorithm

Blondie24 is based on a coevolutionary, evolutionary programming algorithm with self adaptive Gaussian random mutation and 30 individuals. For this study, evolutionary programming with

uniform and Gaussian random noise respectively have been tested along with different PSO-based techniques. Self adaption has not been applied.

In this study different experiments, each using neural networks of a different size, were used to test the impact of neural network size. Experiments with different population and swarm sizes were also tested. Self adaptation was not applied in this study.

3.4 Conclusion

This chapter introduced a general framework for training game playing agents. In the following chapters this framework is applied to the games of tic-tac-toe and checkers.

Chapter 4

Tic-Tac-Toe

This chapter describes the game of tic-tac-toe, and overviews existing techniques for implementing tic-tac-toe game playing agents. It is then shown how the coevolutionary approaches can be used to train tic-tac-toe agents.

4.1 Introduction

Tic-tac-toe is the simplest two-player board game that meets the constraints (two-player, zero sum, perfect knowledge, turn-based, board games) for this study. Although it is not very interesting (it has been solved), tic-tac-toe provides a simple test bed to test new techniques without having to make a large investment in terms of implementation or computer training time.

The goal of this chapter is to explore new techniques for training two-player game agents on tic-tac-toe. In particular, the objective is to adapt existing evolutionary approaches for game-playing agents to be applicable to tic-tac-toe. A new technique is introduced whereby a tic-tac-toe agent is trained with a PSO approach. Different variations of the PSO algorithm are explored and compared to each other as well as the evolutionary programming approach.

Section 4.2 gives a short introduction to the game of tic-tac-toe. Section 4.3 discusses the

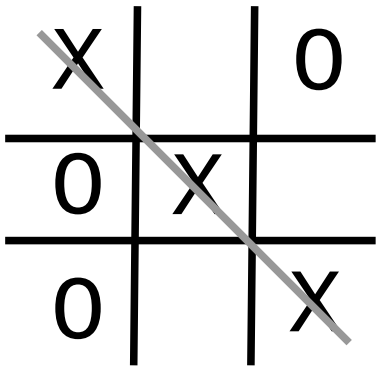

Figure 4.1: Example of Tic-Tac-Toe

training strategies covered in this study, while section 4.5 gives an in-depth coverage of the experimental results. Finally, section 4.6 summarizes the conclusions made from these results.

4.2 The Game of Tic-Tac-Toe

This section overviews the original form of tic-tac-toe. Other tic-tac-toe variations, such as three-dimensional tic-tac-toe are beyond the scope of this thesis. Traditional tic-tac-toe is a game played on a two-dimensional 3x3 grid. Two players take turns to place a mark on the game board. The first player always places an "X" on an unoccupied position on the board and the opponent always places an "O" on an unoccupied position on the board. The game ends when a player obtains a line of three of his markings next to each other (horizontal, vertical or diagonal). This player is declared as the winner. When all the available spaces have been filled without any player having a line of three markings, a draw is declared.

Figure 4.1 illustrates an example board state for which the first player has won.

4.2.1 Comparison between Tic-Tac-Toe and Other Games

This section compares tic-tac-toe to other two-player games. The objective is to compare tic-tac-toe to more complex games, as well as to discuss the merit of examining tic-tac-toe.

The game of tic-tac-toe falls into the same category as board games as chess, checkers, and go. It is a two-player, turn-based game. Tic-Tac-Toe is also zero sum and perfect knowledge, as is the case for checkers, chess, and go.

Tic-Tac-Toe has a relatively small game tree when compared to other mainstream games – the entire game tree consists of an upper limit of just 9!, or 362880 nodes. This makes it possible to solve tic-tac-toe, or in other words to create a perfect player. Because a perfect player can be created, it is easy to define a goal for tic-tac-toe payers. The goal is to become the perfect player. The performance of a tic-tac-toe agent can therefore be measured more accurately than for other games.

Although the search space of tic-tac-toe is relatively tiny, it shares some key characteristics with the very complex game of Go that are not found in other board games such as checkers or chess. These characteristics make it a good test bed for techniques that will not necessarily work for checkers and chess, but can later be applied to go. Some of these characteristics are outlined below:

1. As in go, all pieces in tic-tac-toe have the same value. The value of the pieces does not change throughout the game. In chess and checkers, there are different values assigned to pieces and the value of a piece may change through the course of the game. For example, a checker becomes a king and in chess a passed pawn (a pawn that reached the back row of the opponent) becomes a queen [64].

2. For both tic-tac-toe and go, pieces are placed incrementally on unoccupied spaces on the board in turn, whereas in checkers and chess players start off with a fixed number of pieces.

3. Material advantage is the difference between the number of pieces of the players. Because there is no difference in the number of pieces placed by each player in go and tic-tac-toe, the material advantage is of very little value when evaluating a position. In contrast, a material advantage in chess or checkers usually leads to a win. The minor material

differences that do occur in go, due to handicap stones or prisoners, are usually not a good indication of the winner.

4. Once the pieces of tic-tac-toe or go have been placed on the board they do not move. In chess and checkers moves are made by moving pieces around.

The coevolutionary framework discussed in chapter 3 is applied to tic-tac-toe in this chapter. The results of this study lay the groundwork for algorithms that can be applied to more complex games, such as checkers (refer to chapter 5). Ultimately, these techniques may also be useful to the game of Go, which thus far does not have an agent that can compete on a professional level.

4.3 Training Strategies

This section briefly covers contemporary strategies to play tic-tac-toe. Later in the section learning strategies based on an evolutionary approach and PSO algorithms are introduced.

4.3.1 Introduction

To play tic-tac-toe, a standard minimax tree is a simple and effective strategy – the computer agent expands the current game state to form a game tree in order to calculate the best move. Because the game tree is small, it is even possible to expand the entire game tree. An evaluation function for tic-tac-toe is easily implemented, for it is possible to define the goal of tic-tac-toe (to create rows, columns or diagonals of three tokens) in a simple function.

To improve the time performance of a minimax tree search, an alpha-beta tree may be used to compute the next move. Although an alpha-beta tree finds the solution in less time, the play strength will be the same as for a minimax search – provided that the same evaluation function is used.

For the standard tree approaches mentioned previously, the play strength of the tic-tac-toe agent is related directly to the programming skills and game knowledge of the person developing the game. There is no chance for the agent to discover features of the game without the programmer knowing about the features and adding it to the algorithm. To improve on this drawback researchers have started to implement learning algorithms to train agents to play two-player games [27].

Section 4.3.3 introduces a new training strategy, based on a PSO algorithm to train the neural networks. Different variations of PSO are used in an attempt to find an optimal PSO training strategy for tic-tac-toe.

The results for different combinations of PSO parameters are given in section 4.5. Not only are the results of different strategies compared, but also different combinations of control parameters for each strategy.

4.3.2 Evolutionary Programming Training

This section shows how coevolutionary game-playing algorithms [27] can be adapted for the game of tic-tac-toe. For the random mutation of neural networks both uniform and Gaussian random mutations were tested in this study. Varying population sizes and numbers of hidden units have been used for the experiments in section 4.5.

With reference to chapter 3, the coevolutionary training framework consists of three components. For tic-tac-toe these components are:

1. A standard three-layer feed-forward neural network is used for the static evaluation function. The input layer consists of nine inputs to represent the state of the board and an additional input to identify the player which has to make the next move. One output unit is used to determine the desirability of the input board position. Sigmoid activation functions are used in the hidden and output layers.

2. The evolutionary programming approach as discussed in section 3.2.1 is used for training.

4.3.3 PSO Training Algorithm

This section discusses the changes to the evolutionary programming approach discussed in section 4.3.2 to rather use PSO to train the neural networks.

The training process of the evolutionary algorithm is basically a directed random search. Using PSO, individuals use both their own experience and socially exchanged information to direct the search. It has been shown that the social component of a PSO search will often outperform a random evolutionary search [96].

The components of the PSO approach are similar to that of the evolutionary programming approach, with the only difference in the training method. Neural networks are trained using the following versions of PSO: *gbest* PSO, *lbest* PSO, and an *lbest* GCPSO. In this thesis, a unique implementation of *lbest* GCPSO was developed to take advantage of the increased diversity caused by using smaller neighborhoods.

As discussed in chapter 3, a competition pool consisting of particles and personal best solutions is used. Neural networks therefore compete against current neural networks and previously found best neural networks.

For the tic-tac-toe implementations the inertia weights and acceleration constants were initialized to 1.0. These values were chosen in order to get a good rate of convergence, while still resulting in a wide enough search to learn the game of tic-tac-toe. These values were chosen so that $w > 0.5(c_1 + c_2) - 1$, as shown by Van Den Bergh [96].

Training was stopped after 500 iterations, which proved to be enough time-steps for convergence to take place.

4.4 Performance Measure

In order to fairly compare the performance of different training strategies, there must be an effective measuring process. This section introduces a new measuring criterion for the different tic-tac-toe playing agents. This performance measure is used to provide an objective

and reproducible result.

The play strength of a game playing agent can be measured by playing against human opponents. Unfortunately, playing against humans is problematic: Blondie24's tests against human subjects show that this is a very time-consuming process. Furthermore, the dark side of human nature showed up in some of the games, where a sore loser would terminate the game without resigning or completing the game. Incidentally, the name Blondie24 was the screen-name of a very charming fictional character, in order to persuade opponents to show better manners [27].

Another problem with human opponents is that the results are not objective and repeatable. Humans may have "off days" and it is not possible to reproduce results over time. The playing strength of humans changes on a daily basis.

It is important to note that the measuring criterion (human matches in the case of Blondie24 and computer matches for this study) is only used to determine the final strength of the game-playing agent. The performance criterion is never used to influence the training process.

The learning ability of a neural network tic-tac-toe player is tested against a random tic-tac-toe player. A random player randomly picks any available position and makes a move on that position. This approach was chosen because the playing strength of a random player should be consistent with the probabilities for winning in the standard tic-tac-toe game tree. It can be accurately predicted how well a random player will play: Over a large enough sample a random player will always deliver a consistent result.

The probabilities of outcomes using the entire tic-tac-toe game tree are given in table 4.1. To test the accuracy of the random test player, a sample of 10000 random games are taken and the probability of the different outcomes calculated as given in table 4.2. This shows that a random player over a sample of 10000 games will yield a winning probability very close to that of the actual tic-tac-toe game tree, ensuring a fairly consistent outcome.

In order to evaluate the performance of a neural network player, each neural network

Table 4.1: Calculated Probabilities for Complete Tic-Tac-Toe Game Tree

Draw	0.127
Win as Player 1	0.585
Win as Player 2	0.288

Table 4.2: Probabilities for Random Players

Draw	0.126
Win as Player 1	0.587
Win as Player 2	0.287

player is matched in turn in 10000 games as Player 1 and thereafter in 10000 games as Player 2 against the random player.

The final play strength of a neural network player is then measured using

$$S = (w_1 - 0.585) + (w_2 - 0.288) \tag{4.1}$$

where S is a value between 0 and 1, denoting the play strength of the neural network player, w_1 is the ratio for winning as Player 1 and w_2 is the ratio of winning games as Player 2. In equation (4.1), 0.585 is the probability to win against a random player as Player 1 according to the complete tic-tac-toe game tree. Similarly, 0.288 is the probability to win against a random player as Player 2 (refer to table 4.1).

Stronger players will achieve higher values for S, while values close to 0 indicate weak players. From equation (4.1), negative values may be obtained, which reflects "negative" learning. Negative learning occurs when a player learns to make bad moves rather than winning moves. In the simulations conducted for this study, negative learning did not occur.

A confidence interval is calculated for S using

$$S \pm z_{\alpha/2} \times \frac{\hat{\sigma}}{\sqrt{n}} \tag{4.2}$$

where $z_{\alpha/2}$ is such that $P[Z \geq z_{\alpha/2}]$ with $Z \sim N(0,1)$ and α is the confidence coefficient. For this study the confidence coefficient was chosen to be 90% or $(1-\alpha) = 0.90$.

In equation (4.2), $\hat{\sigma}$ is the standard deviation for a given agent, calculated as

$$\hat{\sigma} = \sqrt{\hat{\pi}_1(1-\hat{\pi}_1) + \hat{\pi}_2(1-\hat{\pi}_2) + 2\hat{\sigma}_{12}} \tag{4.3}$$

where π_1 and π_2 are the probability to win as player one and player two respectively – computed as the number of wins divided by the number of games played; $\hat{\sigma}_{12}$ is the covariance defined as

$$\hat{\sigma}_{12} = \frac{1}{n-1}\left(\sum_{i=1}^{n}(x_{1i}x_{2i}) - \frac{\left(\sum_{i=1}^{n} x_{1i}\right)\left(\sum_{i=1}^{n} x_{2i}\right)}{n}\right) \tag{4.4}$$

where n is the number of games played, and x_{1i} and x_{2i} are defined as the result of the ith game played as Player 1 and Player 2 respectively. Values of x_{1i} and x_{2i} are 1 for a win and 0 for a loss or draw.

4.5 Experimental Results

The experimental results of the evolutionary programming approaches as well as the PSO approaches for tic-tac-toe are evaluated and compared in this section. Section 4.5.1 overviews the experimental procedure. Section 4.5.2 summarizes the obtained play strengths of different implementations as measured by the criterion introduced in section 4.4. A comparison of results is given in section 4.5.3, while the convergence properties of the different implementations are discussed in section 4.5.4.

4.5.1 Experimental Procedure

For each training strategy (i.e evolutionary programming or PSO variation) a number of different experiments were conducted. Experiments were done with different values for the number of neurons in the hidden layer as well as for different numbers of individuals/particles in the population/swarm.

Each experiment was tested with 30 distinct and randomly initialized simulations. The result of each of the 30 simulations was tested against a random player for 10000 games as Player 1 and 10000 games as Player 2 – totaling 300000 tic-tac-toe games on each side. The values in the tables in the following sections are the average results of each simulation's 30 experiments, along with confidence intervals.

For PSO experiments the control parameters were $w = 1$ and $c_1 = c_2 = 1$. The maximum velocity for PSO experiments was capped at 0.1.

For experiments with uniform mutation, $U(0, 1)$ was used, while $N(0, 1)$ was applied to experiments with Gaussian random mutation.

4.5.2 Play Strength

This section summarizes and discusses the performance of the different training approaches using the equation for play strength, defined in equation (4.1).

As explained in chapter 3, the training times of PSO experiments were roughly double than the evolutionary programming approach. This was due to the larger competition pool for the PSO approaches.

Evolutionary Programming Approach (Uniform Mutation)

Table 4.3 summarizes the play strengths for the evolutionary programming approach with uniform random mutation. The evolutionary algorithm managed consistently to outperform a random player, suggesting that the algorithm was able to learn something about the process

of playing tic-tac-toe. This also confirms previous attempts for game-playing agents, showing that the evolutionary process can learn the dynamics of a game [27]. As expected, performance improves with an increase in the number of individuals that are used. This is due to the fact that a larger area of the search space is covered using a larger number of individuals in the population. Very large numbers of individuals do, however, become impractical because of increased time complexity. It is therefore more feasible to try to improve the effectiveness of the learning process than to increase the number of individuals or the number of generations. Neural networks with a smaller number of hidden units generally outperformed larger networks. Larger neural networks (more hidden units) have been shown to overfit training data and perform worse in the general case [101].

From table 4.3, the evolutionary programming approach performed, on average, the best for 7 hidden units, while the average performance for 50 individuals yielded the best performance.

Evolutionary Programming Approach (Gaussian Mutation)

Table 4.4 summarizes the performance of the evolutionary programming approach with Gaussian mutation. The evolutionary program with Gaussian mutation consistently outperformed a random player and showed good performance as defined in equation (4.1). The performance was also better than the evolutionary programming algorithm with uniform random mutations.

As in the previous experiment, the performance improves with an increase in the number of individuals that take part in training, due to the fact that a larger area of the search space is covered. Neural networks with a smaller number of hidden units generally outperformed larger networks. From table 4.4, the evolutionary programming approach with Gaussian mutation performed the best (on average) for 3 hidden units. The best average performance was achieved for 45 individuals in the population.

Global Best Particle Swarm Optimization

Table 4.5 summarizes the performance of the *gbest* PSO approach. The *gbest* PSO outperformed the evolutionary programming approach with uniform random mutations in almost every instance, while not performing as well as evolutionary programming with Gaussian mutations. *Gbest* PSO relies heavily on a uniform random component and only incorporates a limited set of shared knowledge between particles through the single global best particle. The best average performance was obtained for 13 hidden units, while on average, 35 particles provided the best results. The improved performance, compared to the evolutionary programming approach with uniform random mutations, can be attributed to the fact that the particles "share" experiential knowledge.

Local Best Particle Swarm Optimization

Since *lbest* PSO improves diversity, as discussed in section 2.7.4, it is expected that *lbest* should provide better results than *gbest*. This is illustrated in table 4.6 for larger swarm sizes. For swarms of 20 and more particles, the *lbest* PSO outperformed evolutionary programming and *gbest* PSO. For swarms with less than 20 particles the performance is comparable, since there will be less neighborhoods and therefore less exchange of experiential knowledge takes place compared to larger swarm sizes. For swarms with 5 particles *lbest* PSO is essentially equivalent to *gbest* PSO. The *lbest* PSO performed on average best for 50 particles and for 3 hidden units.

Local Best Guaranteed Convergence Particle Swarm Optimization

As explained in chapter 2, GCPSO addresses a drawback of the standard PSO which may cause the standard PSO to converge on a suboptimal solution. The results in table 4.7 show that the *lbest* GCPSO algorithm outperformed both *lbest* and *gbest* in all aspects, and also improved significantly on the results of the evolutionary programming approaches. Even for a swarm size of 25 particles, GCPSO performed better than *lbest* PSO with 50 particles. This causes a substantial reduction in computational time to reach the same performance levels

when using GCPSO. The best average performance was achieved for 3 hidden units and for 45 particles.

4.5.3 Comparison of Results

To compare the relative performance of the PSO approaches to the evolutionary programming approaches, it has to be taken into account that the PSO method uses a training pool that is double the size of the evolutionary programming approaches (due to the inclusion of the personal best positions). For a performance comparison where the same number of competitors are used in the competition pool, refer to table 4.8. This table shows the number of individuals and particles used, as well as the relative performance of the relevant algorithms.

For situations 1 and 2, where smaller numbers of particles and individuals were used, there is relatively little difference in performance between the evolutionary programming and PSO approaches. In situation 1 the *lbest* GCPSO was the best performing algorithm, while in situation 2 evolutionary programming with a Gaussian mutation performed the best. The weakest performers for situations 1 and 2 were *lbest* PSO and evolutionary programming with a uniform random mutation respectively.

Situations 3, 4 and 5, show that the basic *lbest* PSO player outperformed the evolutionary programming with uniform mutation algorithm, but performed weaker than the evolutionary programming with Gaussian mutation algorithm. For situations 3, 4, and 5 the *lbest* GCPSO algorithm consistently showed the best performance. *Gbest* PSO performed weaker than the other PSO variations and always weaker than evolutionary programming with Gaussian mutation. For 4 out of 5 situations *lbest* GCPSO was the best performer.

Refer to figures 4.2 and 4.3 for a visual comparison of the performance for different numbers of hidden unit and individuals/particles respectively. In each case the average performance for the number if hidden units or number of particles are used. These figures clearly illustrate the differentiation between the evolutionary programming and PSO algorithms.

Table 4.3: Tic-Tac-Toe Performance Results for the Evolutionary Programming Approach (Uniform Mutation)

	Hidden units						
Size	**3**	**5**	**7**	**9**	**11**	**13**	**Average**
5	0.0625 ±0.00202	0.07887 ±0.00201	0.02678 ±0.00204	0.06646 ±0.002	0.05872 ±0.00201	0.03488 ±0.00203	*0.0547*
10	0.07534 ±0.002	0.06766 ±0.00204	0.08151 ±0.00203	0.06 ±0.002	0.02857 ±0.00202	0.02723 ±0.002	*0.05672*
15	0.06208 ±0.00203	0.0638 ±0.00204	0.05476 ±0.00202	0.04459 ±0.00199	0.06152 ±0.00202	0.03997 ±0.002	*0.05445*
20	0.05555 ±0.00202	0.06137 ±0.00204	0.07714 ±0.002	0.04829 ±0.00195	0.07215 ±0.00201	0.05463 ±0.00199	*0.06152*
25	0.02155 ±0.00199	0.06987 ±0.00198	0.08327 ±0.00197	0.07519 ±0.00199	0.04687 ±0.00199	0.06109 ±0.00198	*0.05964*
30	0.08455 ±0.00197	0.0398 ±0.00196	0.09754 ±0.00197	0.09805 ±0.00203	0.04744 ±0.00199	0.04383 ±0.00201	*0.06853*
35	0.07458 ±0.002	0.09637 ±0.002	0.13902 ±0.00195	0.12568 ±0.00199	0.05659 ±0.00196	0.06302 ±0.00197	*0.09254*
40	0.10644 ±0.002	0.05649 ±0.00196	0.10075 ±0.00197	0.10338 ±0.00201	0.08699 ±0.00195	0.0579 ±0.00197	*0.08532*
45	0.09582 ±0.00196	0.07996 ±0.00201	0.10357 ±0.00197	0.11067 ±0.00203	0.08293 ±0.00197	0.10664 ±0.00198	*0.0966*
50	0.129 ±0.00201	0.08495 ±0.00201	0.12824 ±0.00197	0.10934 ±0.002	0.08866 ±0.00198	0.10615 ±0.00198	*0.10772*
Average	*0.07674*	*0.06991*	*0.08926*	*0.08416*	*0.06305*	*0.05953*	

Table 4.4: Performance Results for the Evolutionary Approach (Gaussian Mutation)

	3	**5**	**7**	**9**	**11**	**13**	**avg**
5	0.06269 ±0.0020	0.0809 ±0.0020	0.02857 ±0.0020	0.06649 ±0.0020	0.05853 ±0.0020	0.03502 ±0.0020	*0.05537*
10	0.0746 ±0.0020	0.07172 ±0.0020	0.08101 ±0.0020	0.06004 ±0.0020	0.02811 ±0.0020	0.02567 ±0.0020	*0.05686*
15	0.06333 ±0.0020	0.06615 ±0.0020	0.05613 ±0.0020	0.04448 ±0.0020	0.06092 ±0.0020	0.03997 ±0.0020	*0.05516*
20	0.15041 ±0.0020	0.18642 ±0.0020	0.15238 ±0.0020	0.15228 ±0.0020	0.16596 ±0.0020	0.13596 ±0.0020	*0.15724*
25	0.11189 ±0.0020	0.12674 ±0.0020	0.09555 ±0.0020	0.14377 ±0.0020	0.13588 ±0.0020	0.08699 ±0.0020	*0.1168*
30	0.14417 ±0.0020	0.1318 ±0.0020	0.07485 ±0.0020	0.09998 ±0.0020	0.10464 ±0.0020	0.12981 ±0.0020	*0.11421*
35	0.15065 ±0.0020	0.14817 ±0.0020	0.12498 ±0.0020	0.08403 ±0.0020	0.13163 ±0.0021	0.09628 ±0.0020	*0.12262*
40	0.11712 ±0.0020	0.09647 ±0.0020	0.12466 ±0.0020	0.14957 ±0.0021	0.13067 ±0.0020	0.15743 ±0.0020	*0.12932*
45	0.18628 ±0.0020	0.13226 ±0.0020	0.18765 ±0.0020	0.14246 ±0.0020	0.15379 ±0.0021	0.1716 ±0.0020	*0.16234*
50	0.12674 ±0.0020	0.13231 ±0.0021	0.1243 ±0.0020	0.13683 ±0.0020	0.16497 ±0.0020	0.1502 ±0.0021	*0.13923*
avg	*0.11879*	*0.11729*	*0.10501*	*0.10799*	*0.11351*	*0.10289*	

Table 4.5: Tic-Tac-Toe Performance Results for Global Best PSO

	Hidden units						
Particles	**3**	**5**	**7**	**9**	**11**	**13**	**Average**
5	0.05918 ±0.00196	0.03449 ±0.002	0.0556 ±0.00198	0.0708 ±0.00198	0.06619 ±0.00201	0.08851 ±0.00202	*0.06246*
10	0.07498 ±0.00203	0.04014 ±0.002	0.10841 ±0.00204	0.03457 ±0.00205	0.08992 ±0.00204	0.08273 ±0.00199	*0.07179*
15	0.09788 ±0.00201	0.13255 ±0.00196	0.10006 ±0.00198	0.12311 ±0.00201	0.08614 ±0.00203	0.08543 ±0.00195	*0.1042*
20	0.09953 ±0.00203	0.06293 ±0.002	0.07664 ±0.00199	0.06835 ±0.00201	0.10062 ±0.00198	0.11344 ±0.00203	*0.08692*
25	0.10861 ±0.00197	0.07213 ±0.00199	0.10167 ±0.00203	0.06894 ±0.00201	0.06228 ±0.002	0.08952 ±0.00203	*0.08386*
30	0.04794 ±0.00202	0.10676 ±0.00203	0.0532 ±0.00201	0.12868 ±0.00201	0.12557 ±0.00201	0.11616 ±0.00202	*0.09638*
35	0.14248 ±0.00204	0.11725 ±0.00199	0.11247 ±0.00199	0.11619 ±0.00201	0.11053 ±0.00205	0.14262 ±0.00201	*0.12359*
40	0.10342 ±0.002	0.08186 ±0.00198	0.09699 ±0.002	0.12882 ±0.00201	0.16638 ±0.00202	0.11037 ±0.00198	*0.11464*
45	0.13621 ±0.00199	0.11999 ±0.002	0.12102 ±0.00203	0.09327 ±0.00202	0.07616 ±0.00201	0.11382 ±0.00205	*0.11008*
50	0.12875 ±0.00201	0.16687 ±0.00201	0.11214 ±0.00201	0.06322 ±0.00202	0.0938 ±0.00202	0.11466 ±0.00201	*0.11324*
Average	*0.0999*	*0.0935*	*0.09382*	*0.08959*	*0.09776*	*0.10573*	

Table 4.6: Tic-Tac-Toe Performance Results for Local Best PSO

	Hidden units						
Particles	**3**	**5**	**7**	**9**	**11**	**13**	**Average**
5	0.07858 ±0.00197	0.06875 ±0.00198	0.04979 ±0.002	0.04468 ±0.00199	0.03065 ±0.00203	0.0679 ±0.002	*0.05672*
10	0.05305 ±0.00201	0.05304 ±0.00199	0.06151 ±0.00202	0.11203 ±0.00199	0.06062 ±0.00204	0.05339 ±0.00203	*0.06561*
15	0.1176 ±0.002	0.0781 ±0.00201	0.13297 ±0.00201	0.07717 ±0.00198	0.05202 ±0.00203	0.06546 ±0.00204	*0.08722*
20	0.10191 ±0.00202	0.12739 ±0.00202	0.12365 ±0.002	0.12648 ±0.00199	0.09673 ±0.00201	0.10506 ±0.00201	*0.11354*
25	0.13577 ±0.00199	0.15401 ±0.00199	0.10648 ±0.00203	0.13214 ±0.00201	0.10005 ±0.00199	0.10925 ±0.00201	*0.12295*
30	0.19244 ±0.002	0.14969 ±0.00201	0.11458 ±0.002	0.19765 ±0.00201	0.19501 ±0.00202	0.19442 ±0.002	*0.17397*
35	0.17926 ±0.00202	0.16741 ±0.00198	0.19355 ±0.00201	0.10946 ±0.00202	0.1901 ±0.00198	0.15123 ±0.00197	*0.16517*
40	0.19117 ±0.00199	0.1799 ±0.00198	0.16313 ±0.00201	0.19634 ±0.00194	0.20092 ±0.00203	0.13666 ±0.00201	*0.17802*
45	0.19107 ±0.002	0.22856 ±0.00199	0.18327 ±0.00198	0.20869 ±0.00196	0.16469 ±0.00199	0.19915 ±0.00198	*0.19591*
50	0.18288 ±0.00201	0.20141 ±0.00198	0.19735 ±0.00199	0.1912 ±0.002	0.24944 ±0.00196	0.16355 ±0.00196	*0.19764*
Average	*0.14237*	*0.14083*	*0.13263*	*0.13958*	*0.13402*	*0.12461*	

Table 4.7: Tic-Tac-Toe Performance Results for Local Best GCPSO

	Hidden units						
Particles	**3**	**5**	**7**	**9**	**11**	**13**	**Average**
5	0.11045 ±0.00204	0.05261 ±0.00198	0.05004 ±0.00201	0.05118 ±0.00201	0.03005 ±0.00202	0.08309 ±0.00195	*0.0629*
10	0.13816 ±0.00197	0.1317 ±0.00202	0.16691 ±0.00205	0.14373 ±0.00202	0.08881 ±0.00203	0.07082 ±0.00201	*0.12335*
15	0.1873 ±0.00198	0.17595 ±0.00202	0.1552 ±0.00201	0.13728 ±0.00197	0.1331 ±0.002	0.15411 ±0.00203	*0.15716*
20	0.21622 ±0.00197	0.16008 ±0.00198	0.16569 ±0.00202	0.19986 ±0.00198	0.14171 ±0.00201	0.17144 ±0.00196	*0.17583*
25	0.2094 ±0.00201	0.17489 ±0.002	0.19025 ±0.002	0.20558 ±0.00197	0.22612 ±0.00195	0.21332 ±0.00196	*0.20326*
30	0.22684 ±0.00197	0.22142 ±0.00195	0.22186 ±0.00195	0.19961 ±0.00198	0.17192 ±0.00194	0.18145 ±0.00196	*0.20385*
35	0.25182 ±0.00193	0.22693 ±0.002	0.26493 ±0.00194	0.20511 ±0.00198	0.23813 ±0.00195	0.19218 ±0.00197	*0.22985*
40	0.22233 ±0.00193	0.26152 ±0.00196	0.23628 ±0.00195	0.22672 ±0.00201	0.22803 ±0.00201	0.24134 ±0.00196	*0.23604*
45	0.24962 ±0.00195	0.27315 ±0.00196	0.2602 ±0.00195	0.23681 ±0.00195	0.26781 ±0.00194	0.22474 ±0.00196	*0.25206*
50	0.24166 ±0.00198	0.22015 ±0.00196	0.25087 ±0.00197	0.18681 ±0.00196	0.23252 ±0.00201	0.2407 ±0.00198	*0.22879*
avg	*0.20538*	*0.18984*	*0.19622*	*0.17927*	*0.17582*	*0.17732*	

Table 4.8: Tic-Tac-Toe Performance Comparison

Situation	1	2	3	4	5
EP Individuals	10	20	30	40	50
PSO Particles	5	10	15	20	25
Evolutionary (Uniform)	0.05672	0.06152	0.06855	0.08532	0.10772
Evolutionary (Gaussian)	0.05686	0.15724	0.11421	0.12931	0.13923
gbest PSO	0.06246	0.07179	0.01041	0.08692	0.08386
lbest PSO	0.05670	0.06561	0.08722	0.11350	0.12290
lbest GCPSO	0.06290	0.12335	0.15716	0.17580	0.20320

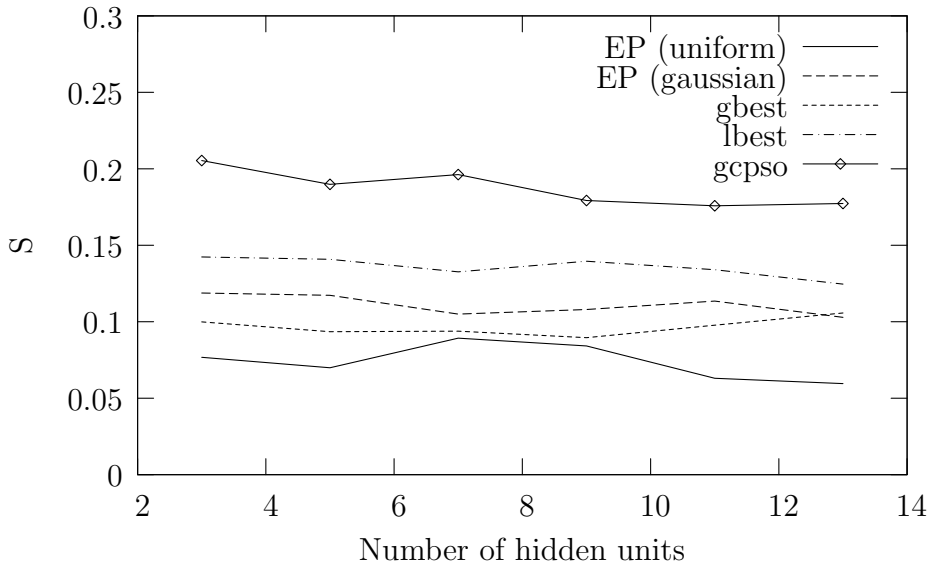

Figure 4.2: Performance Against Hidden Units

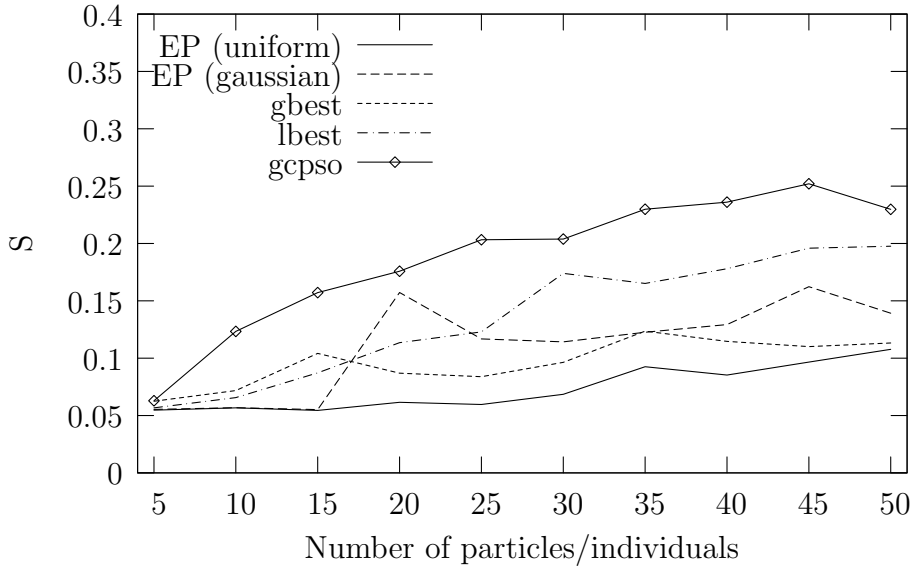

Figure 4.3: Performance Against Swarm/Population Size

4.5.4 Convergence Properties

Convergence measures help to determine if an algorithm has reached a convergence criterion, such as a local optimum. Such measures, plotted as a function of iteration number, also give an indication as to whether an algorithm continues to improve, or if it stagnates. This section presents convergence results for the different algorithms. For this purpose, convergence is measured as the mean of the sum of the squares of the weights of the best individual after each iteration, i.e,

$$c(t) = \frac{\sum_{i=0}^{n-1} w_i^2(t)}{N} \tag{4.5}$$

where $c(t)$ is the convergence measure at iteration t, N is the total number of weights in the neural network, and $w_i(t)$ is the value of the weight at index i. The premise is that when $|c(t) - c(t-1)| \rightarrow 0$, weight adjustments are small, which indicates that little learning takes place.

The experiments with the evolutionary programming approaches (both uniform and Gaussian mutation) can be divided into two classes of convergence behavior. The first behavior is

exhibited by experiments with 15 or fewer individuals, while experiments of larger population sizes fall into the second group. A sample behavior of each group is illustrated in figures 4.4 and 4.5 respectively.

The first class shows that smaller population sizes exhibited an oscillating behavior, with no convergence of the weight values within the 500 generations (refer to figure 4.4). This behavior was observed regardless of the number of hidden units used in the neural network or mutation algorithm.

The next significant observation is that for experiments with 20 or more individuals (refer to figure 4.5), most of these experiments converged. Large fluctuations early in the training process converged fairly rapidly and were followed by a smoother convergence pattern (see figure 4.5). There were, however, a small number of simulations that failed to train anything without any change for almost all iterations. A sample behavior for these experiments is illustrated in figure 4.6. This shows that it is likely to find good training with an evolutionary process, but there exists a risk (albeit a small one) that the process may converge on a weaker than expected solution.

The *gbest* PSO convergence characteristics are very similar to the second group of the evolutionary programming approaches, with the exception that the fluctuations in weight values are smoother. Refer to figure 4.7 for a typical example of *gbest* convergence behavior. For *gbest* PSO, the particles have an inertia weight associated with their flight through n-dimensional space, which serves to smooth the search trajectory. Another point to note is that the two biggest problems that were experienced with the evolutionary programming approaches, i.e, extreme oscillating behavior for small populations (shown in figure 4.4) and premature convergence (shown in figure 4.6), were not experienced with *gbest* PSO. It is important to note that this behavior of the PSO algorithms can only be ensured for good values of the PSO control parameters as explained in chapters 2 and 3.

Figure 4.8 illustrates the typical convergence behavior of the *lbest* PSO. In comparison with the *gbest* PSO behavior shown in figure 4.7, *lbest* PSO resulted in a more volatile search,

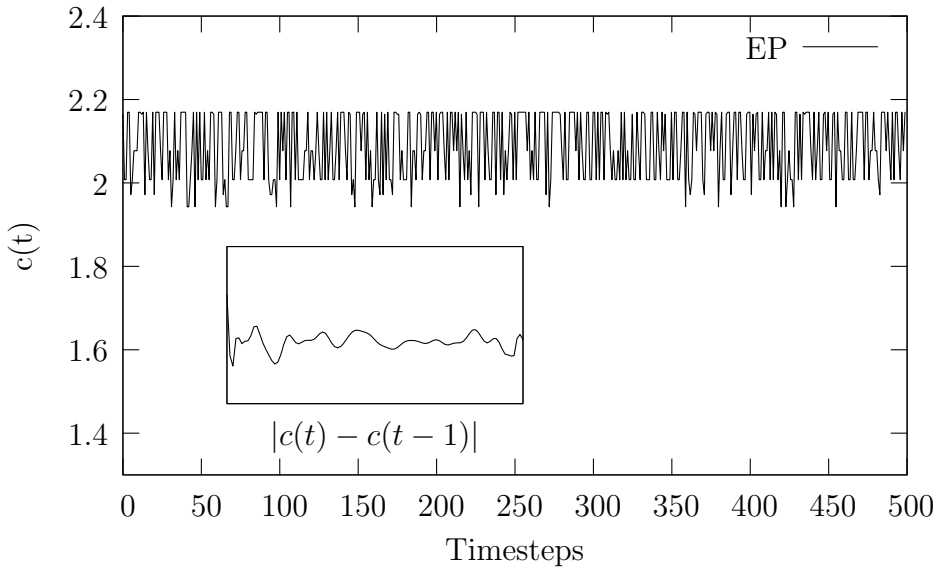

Figure 4.4: Convergence Behavior for Small Population Size for the Evolutionary Programming Approach (both uniform and Gaussian mutation)

indicating that *lbest* PSO searches a larger area of the search space. *Lbest* PSO did converge in the final timesteps with values close to zero for $|c(t) - c(t-1)|$.

Figure 4.9 illustrates the convergence behavior of the *lbest* GCPSO implementation. Initially, large adjustments were made, and then, after approximately 25 iterations, changes to weight values were small. This indicates that the PSO may have converged or stagnated on a local minimum. Through adjustment of the global best solution using equation (2.16) in the case of stagnation, the GCPSO has the ability to escape the local minimum or stagnation shortly after 250 iterations, and to continue the search for a better solution. Because suboptimal convergence was avoided, as shown in 4.9, bigger parts of the search space were covered.

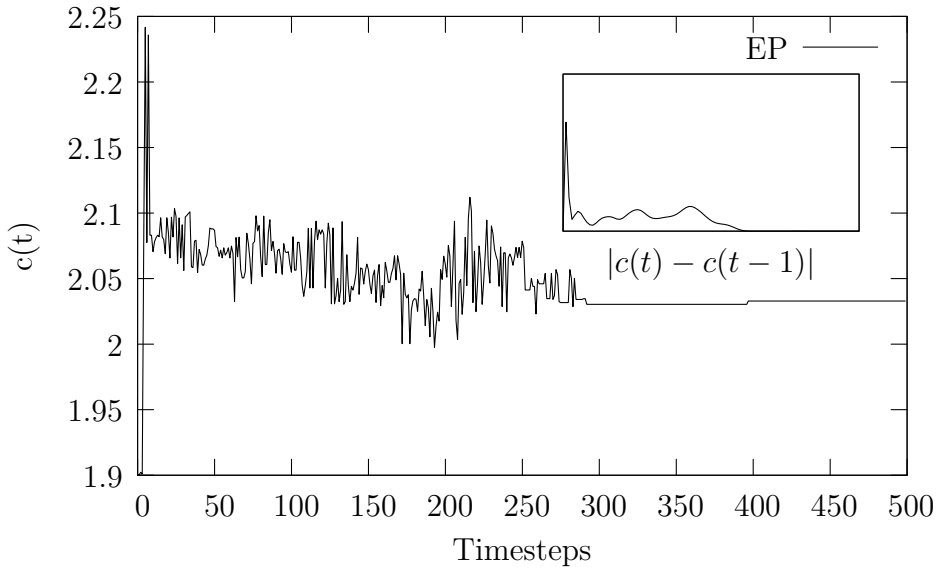

Figure 4.5: Convergence Behavior for Large Population Size for the Evolutionary Programming Approach (both uniform and Gaussian mutation)

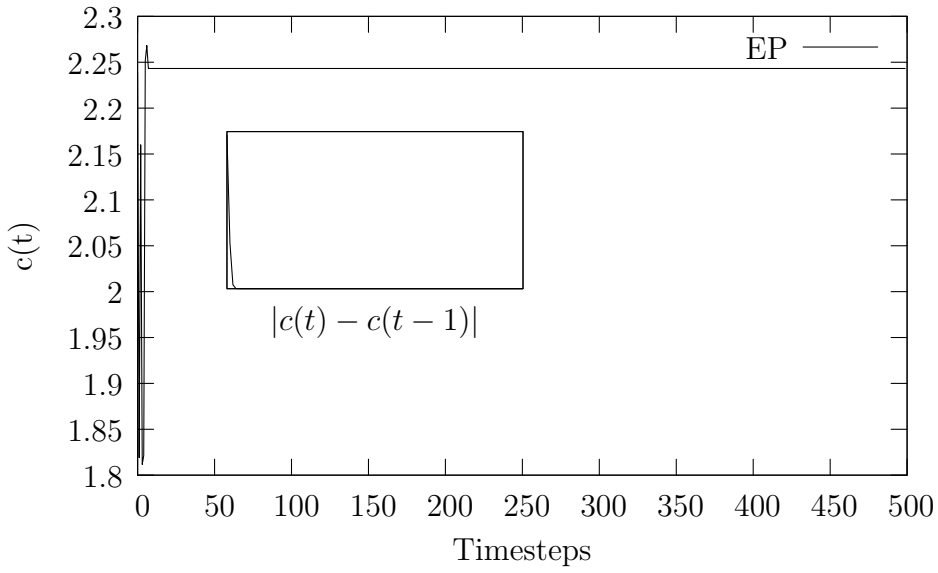

Figure 4.6: Suboptimal Convergence Illustration for the Evolutionary Programming Approach (both uniform and Gaussian mutation)

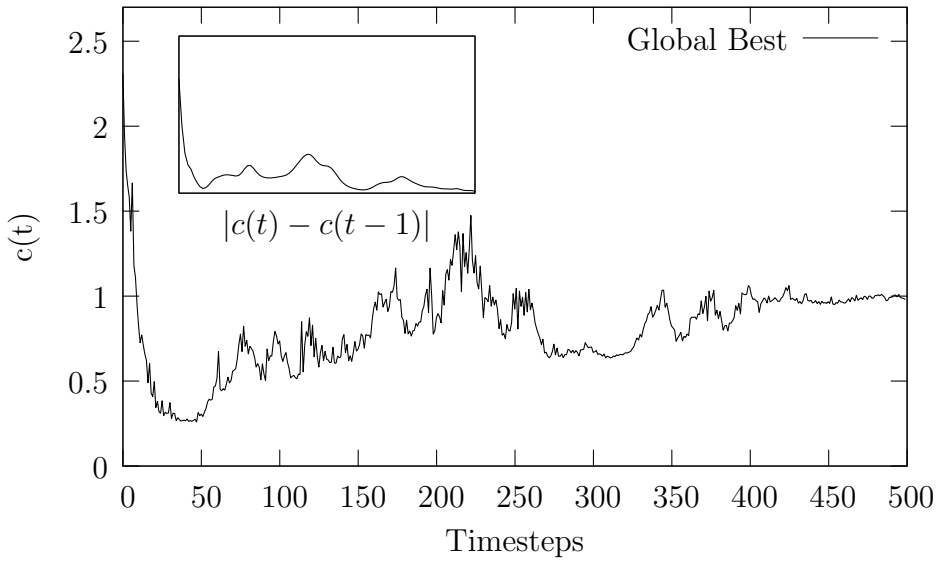

Figure 4.7: Convergence Behavior for the *gbest* PSO

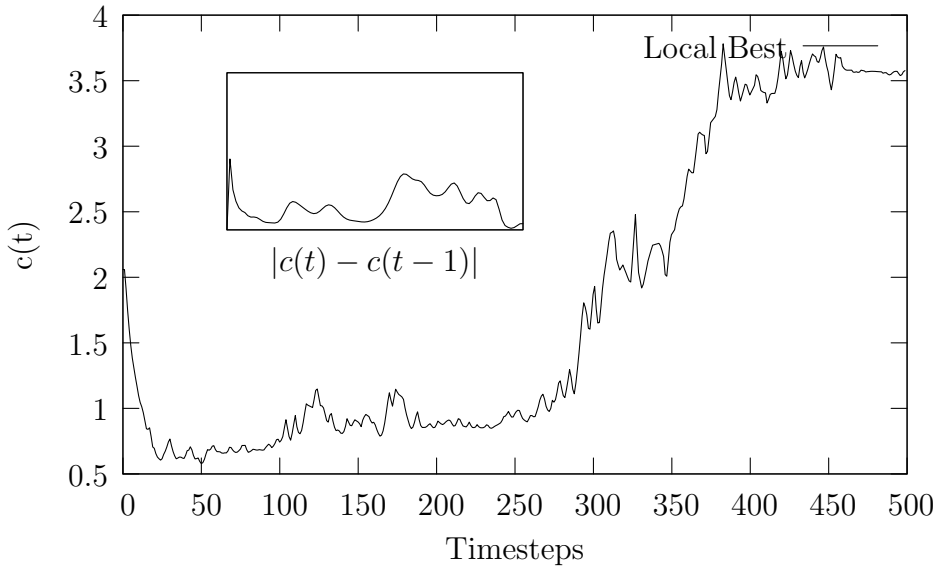

Figure 4.8: Convergence Behavior for the *lbest* PSO

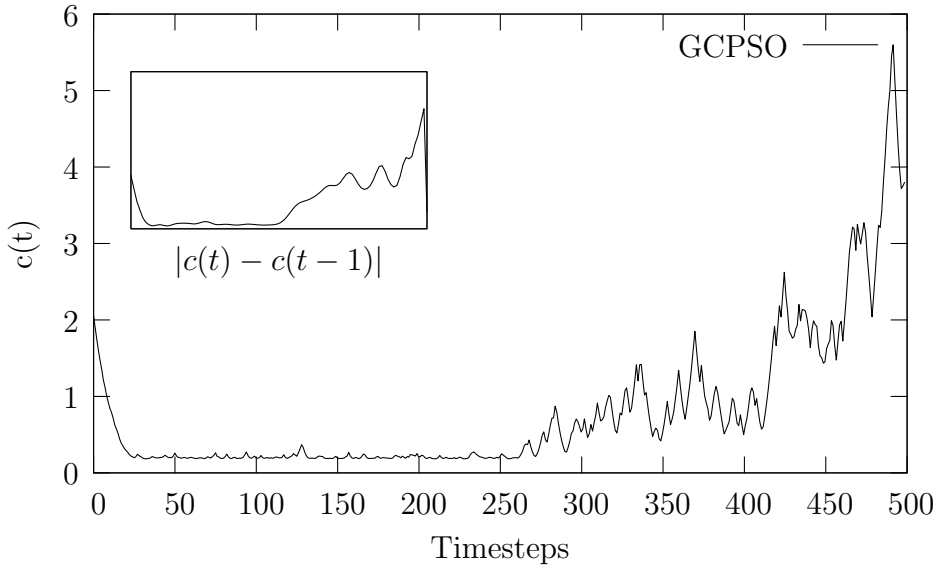

Figure 4.9: Convergence Behavior for the *lbest* GCPSO

4.6 Conclusion

This chapter introduced a new performance measure for two-player game agents, based on competition against a random player. This performance measure is simple to implement, objective, repeatable, and it allows for a quantitative measure for game playing agents.

Other studies have shown that it is possible to train neural networks as game playing agents using evolutionary processes [10]. The results, based on the performance measure introduced in this chapter, confirmed that these neural networks can be trained to play tic-tac-toe. This chapter introduced PSO-based training of neural networks for game-playing agents and used the performance measure to compare the results. PSO game-playing agents were based on PSO *lbest* PSO, *gbest* PSO, and *lbest* GCPSO algorithms. The GCPSO algorithm was based on a local best PSO strategy. *Lbest* GCPSO has not yet been tested in other application areas.

It was shown that evolutionary programming with a Gaussian random mutation performed better than a uniform random mutation and *gbest* PSO. The *lbest* PSO and GCPSO implementations consistently outperformed the evolutionary programming approaches, based on

the random player performance measure.

Gbest PSO was successful in learning to play tic-tac-toe according to the random player performance measure. Furthermore, *gbest* PSO was able to reach a superior play strength to that of the evolutionary programming approach with a uniform random mutation, but failed to reach the performance level of a Gaussian random mutation evolutionary programming algorithm.

The *lbest* PSO increased diversity with its neighborhood implementation. This algorithm was able to gain a better score against a random player. It outperformed both the evolutionary programming algorithms, as well as *gbest* PSO.

The *lbest* GCPSO was shown to be the best performer for tic-tac-toe, as measured by equation (4.1). It managed to learn the game of tic-tac-toe from zero knowledge and outperformed all the other algorithms.

In the next chapter the PSO algorithms that were discussed in this chapter are applied to the game of checkers.

Chapter 5

Checkers

This chapter illustrates how the PSO approach discussed in chapter 2 can be used to train a checkers game-playing agent. Results are compared against the evolutionary programming approaches for checkers game-playing agents as discussed in chapter 3. Evolutionary programming (both Gaussian and uniform mutation) as well as different variations of PSO were studied.

5.1 Introduction

Checkers is a two-player game that is too complex to be solved by modern computing equipment. The search space allows only a limited number of moves to be examined in a practical amount of time. Although checkers is simpler (having a smaller game tree) than chess or go, this game provides enough complexity to push learning agent technology toward its limits [27][84].

This chapter introduces new approaches for learning the game of checkers. Section 5.2 gives a short introduction to the game of checkers. Thereafter different techniques applied to checkers-playing agents are discussed in sections 5.3 and 5.4. Section 5.5 presents the results of the different checkers-playing agents discussed in section 5.2. Section 5.6 concludes this

chapter.

5.2 The Game of Checkers

Checkers is a board game that is played on an eight-by-eight checkered board (refer to figure 5.1 for the checkers board and initial board setup). Each player starts with twelve pieces, where each piece is called a checker. Checkers may only occupy white squares. One player plays with black checkers, and the opponent plays with red (or white in some cases) checkers [74].

The game has relatively simple rules:

1. Checkers move diagonally forward, one block at a time to an unoccupied block (a block that does not have a checker on it). For example, in figure 5.1, the checker on block 21 can only be moved to 17, while the checker on 22 can be moved to either 17 or 18. The checker on 11 can be moved to 15 or 16.

2. If an opposing checker is directly in front of a checker, and the block behind the opposing checker is open, the checker can "jump over" that checker – moving two diagonal blocks forward. The opposing checker that is jumped over is then removed from the board.

3. If a jump is a possible move, the player must make the jump (no other move is allowed). As for normal moves, jumps are only allowed in the forward direction (towards the opponent). There are variations on the forced jump rule, but for the purposes of this study it is assumed that jumps are forced.

4. Jumps can be taken in succession of each other. In figure 5.2 the black checker may jump from 32 to 23 to 14 to 7, removing all the white checkers. When multiple jumps are carried out, all the jumps must be taken by a single checker. This counts as one move by the player. Once again, the rule applied in this study requires that if multiple jumps

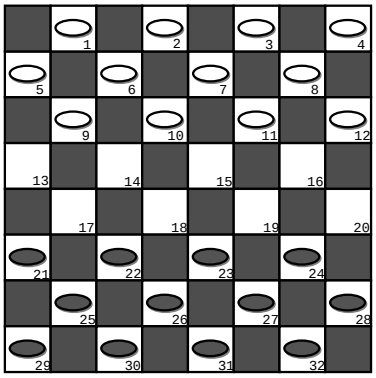

Figure 5.1: Starting Position of Checkers

are possible, they are compulsory. Thus, for figure 5.2 all the jumps must be taken by the black checker.

5. When a checker successfully reaches the back row of the opponent's side of the board, it becomes a king. A king moves in the same way that a checker does, but the restriction of moving forward only is lifted. Kings are allowed to move in any diagonal direction. They can also make multiple jumps as long as the same king makes the multiple jumps in one move.

6. The game terminates when:

   (a) One of the players does not have any legal moves to make – making that player the loser. This usually happens when all his pieces are removed from the board, but it can also happen when all the checkers for that player are trapped.

   (b) When only circular moves are possible, and the game will never end in a win for either player, a draw is declared.

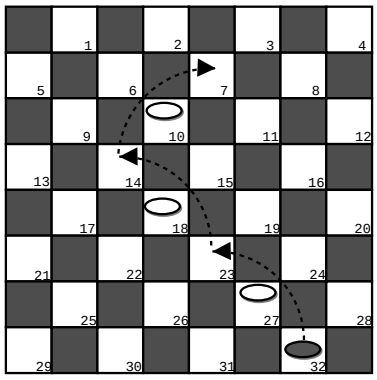

Figure 5.2: Example Position of Checkers

5.3 Training Strategies

This section summarizes the current technology applied to train successful checkers agents. It covers traditional tree search approaches as well as more recent self learning agents such as Blondie24 [27].

Training approaches based on chapter 3 are introduced and evaluated against a random player. The measuring scheme is also based on performance measured against a random player.

5.3.1 Introduction

As with most two-player games, the most successful computer players are based on tree building algorithms. Chinook is the world's most successful checkers player. It is the first computer to win the combined human-computer world championship [84]. The key to Chinook's strength is four-fold:

1. Chinook has an optimal tree building algorithm, which creates very deep tree searches and expands on critical points.

2. An end-game database allows Chinook to perform perfect play for all positions with eight or fewer pieces.

3. An extensive opening database gives Chinook a very strong opening play.

4. Chinook's evaluation function takes two dozen weighted factors into account. This sophisticated evaluation algorithm will consistently choose good positions.

Although Chinook has been shown to be the best checkers player ever, every piece of knowledge embedded within Chinook has been coded by humans. It is not possible for Chinook to gain any knowledge of the game on its own.

At the turn of the century a learning agent for checkers, based on a neural network trained by a coevolutionary programming approach, was introduced [10][27]. This showed that a neural network can be trained to play the game of checkers without a human trainer. In fact, the agent (called Blondie24) is able to beat most humans (it has a rating of 2045, designated as "expert" [28]), even though it started training with nothing more than the rules of the game and the type, location and number of pieces [10][27].

5.3.2 Evolutionary Programming Approach

With reference to the general framework presented in chapter 3, the evolutionary programming approach used in this chapter follows the outline discussed in section 3.2.1. The neural network receives the 32 positions of a checkers board, along with the player to move, as input parameters and yields the desirability of the board with a single output parameter. The output is in the range $(0, 1)$, where 0 denotes a weak or undesirable position for the indicated player, and values closer to 1 show a very strong or desirable position. The neural network is used as the static evaluation function for a game tree.

The evolutionary programming training have been tested for both uniform mutation of neural network weights as well as Gaussian mutation.

5.3.3 Particle Swarm Optimization

Chapter 4 has shown that the *lbest* GCPSO outperformed the evolutionary programming algorithm as training strategy within the coevolutionary game agent training model. In this chapter the *gbest* PSO, *lbest* PSO and *lbest* GCPSO approaches are used as training strategies to determine if the PSO can be applied to learning the game of checkers.

With reference to the three components of the general framework, discussed in chapter 3, a search tree with ply-depth 1 is used. A neural network with the same architecture as for the evolutionary programming approach (discussed above) is used with the PSO-based training algorithms.

5.4 Measuring Performance

This section develops an effective and unbiased criterion to compare the performance of the different algorithms. Similar to the performance criterion used for tic-tac-toe (refer to section 4.4), the criterion for checkers is based on the performance of a random player.

However, to accurately measure the performance of a checkers player is more difficult than that of tic-tac-toe, since it is not possible to construct a complete game tree for checkers in a reasonable amount of time, as can be done for tic-tac-toe. It is not possible to get absolute values for the probabilities for the complete checkers game tree, and only approximate values can be calculated.

It was shown in section 4.4 that it is possible to get a good approximation of the probabilities of reaching a given result in a search tree by matching random players against each other. A statistical confidence interval can also be calculated on the results of random players to ensure that a relatively good estimate has been reached.

To calculate an approximation of the probabilities of the checkers game tree, random players were matched against each other for one million checkers games. The estimates for each possible outcome (i.e. player one win, player two win or draw) was calculated by dividing

the number of times the outcome was reached by the total number of games played, as given in equation (5.1):

$$\hat{\pi}_o = \frac{n_o}{n} \tag{5.1}$$

where $\hat{\pi}_o$ is the estimate for outcome o, n_o is the number of times outcome o was reached and n is the total number of games.

A confidence interval for each outcome is calculated using:

$$\hat{\pi}_o \pm z_{\alpha/2} * \frac{\hat{\sigma}_o}{\sqrt{n}} \tag{5.2}$$

where $\hat{\pi}_o$ is the estimated probability for outcome o and $z_{\alpha/2}$ is such that $P[Z \geq z_{\alpha/2}]$ where $Z \sim N(0,1)$. To calculate the confidence for checkers probabilities, a confidence interval of 95% or $(1-\alpha) = 0.95$ was used. The confidence value for checkers was introduced because the game tree and therefore the absolute outcomes for checkers can't be computed as in section 4.4.

To calculate the standard deviation, $\hat{\sigma}_o$, of outcome o, equation (5.3) is used, where x_i is equal to 1 if the result of game i was outcome o and 0 otherwise:

$$\hat{\sigma}_o = \frac{1}{n-1}\left(\frac{\sum_{i=1}^{N} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}{n}\right) \tag{5.3}$$

The estimated values for the outcomes of the checkers game tree is given in table 5.1. In contrast to tic-tac-toe, the probability for the player that moves first is not significantly higher than the probability for the second player to win. The probability for a draw is very small, which means that random players will seldom draw. This is generally not the case in closely matched, strong (human and computer) players, since the rules of the game do not apparently favor either player as in the case of tic-tac-toe. This explains why draws often occur in practice.

In order to calculate the playing strength of a checkers computer agent, matched against a random player for a series of games, the following equation is used:

Table 5.1: Calculated Probabilities for Checkers Game Tree

Draw	0.005546	±0.0001
Win as Player 1	0.504927	±0.0001
Win as Player 2	0.489527	±0.0001

$$S = w_1 - 0.5 + w_2 - 0.5 \tag{5.4}$$

where S is the play strength of the checkers agent. The values w_1 and w_2 are the ratios of games won as Player 1 and Player 2 respectively. In both cases, 0.5 is subtracted as it is the probability for each player to win the game. A value of 0.5 is used because the probability to win as either player is relatively close to each other, as show in table 5.1.

A confidence interval is calculated for S using

$$S \pm z_{\alpha/2} * \frac{\hat{\sigma}}{\sqrt{n}} \tag{5.5}$$

with $Z \sim N(0,1)$, and α is the confidence coefficient. Refer to section 4.4 for a complete discussion of the performance interval equation.

5.5 Experimental Results

This section presents the results of different strategies for training checkers playing agents. The playing strength of these strategies is compared, and the performance of different combinations of hidden units and swarm sizes is investigated.

Section 5.5.1 gives the experimental procedure while sections 5.5.2 and 5.5.3 summarize the experimental results. This section also shows the results of the play strength defined in section 5.4 for each experiment.

5.5.1 Experimental Procedure

To test the performance of different strategies, a series of experiments was conducted for each strategy. Each experiment was run with different sets of parameter values and/or training strategies. For this study the parameter values tested were the number of hidden units and the number of particles or individuals used.

For each experiment, a total of 30 simulations, running for 500 iterations each and tested against a random player were used to test the checkers agents. Refer to section 4.5.1 for a full breakdown of the experimental procedure.

The 30 runs for smaller experiments (experiments with small neural networks, and a small number of individuals) took about 8 hours to complete. Larger experiments (experiments with large neural networks and a large number of individuals) took upwards of 100 hours to complete the 500 iterations.

For PSO experiments the control parameters were $w = 1$ and $c_1 = c_2 = 1$. These values where chosen so that $w > 0.5(c_1 + c_2) - 1$ holds in order to get convergent behavior [96]. The maximum velocity for PSO experiments was capped at 0.1.

For experiments with uniform mutation, $U(0, 1)$ was used, while $N(0, 1)$ was used for experiments with Gaussian random mutation.

5.5.2 Play Strength

This section compares the performance of the different strategies with reference to equation (5.4).

Evolutionary Algorithm (Uniform Mutation)

This section presents the results for the evolutionary programming process, trained with uniform mutation. The results of the evolutionary algorithm (as calculated from equation (5.4)) are listed in table 5.2. Table 5.2 shows the performance for different numbers of hidden units

and number of individuals.

All the results of the evolutionary trained agents are relatively close to the maximum score 1, with the lowest score being 0.89. These scores are significantly better than the results that are expected for a random player. This shows that evolutionary programming is capable of successfully learning the game of checkers. It does not often lose or draw against a random player. These results are also consistent with other findings [10][27].

As more individuals are added to the population, the performance generally increases. This result is expected, since larger populations will cover a larger area of the search space. Larger neural network structures did not increase the performance, and the performance remained almost unchanged.

The best performance was 0.9661, for a neural network with 5 hidden units and 40 individuals. It was closely followed by the 10 hidden unit neural network for 40 individuals with a score of 0.9614.

The confidence intervals are very small with values of 0.001 or smaller. Small confidence intervals show that all the experiments leaned towards a similar performance and that there are no large deviations in performance.

Evolutionary Algorithm (Gaussian Mutation)

This section presents the results of evolutionary programming using Gaussian mutation. The play strength of the algorithm (as calculated from equation (5.4)) is listed in table 5.3.

All the results of the evolutionary programming approach with Gaussian mutation are relatively high (the lowest score being 0.8331), and do not differ significantly with that of uniform mutations.

As in the case of uniform mutation, the best score for Gaussian mutation was obtained with a relatively large number of populations and a small neural network. The best score of 0.9556 was shown for neural networks with 5 hidden units and populations of 30 individuals.

The performance improved when more individuals were added to the population, while

performance worsened slightly when more hidden units were added to the neural network.

The confidence intervals are very small (in most cases less than 0.0015), indicating that there are no large deviations in performance.

Gbest PSO Approach

This section presents the results of neural networks trained as checkers-playing agents with the *gbest* PSO approach. The *gbest* algorithm for PSO training uses the position of the best particle to distribute knowledge of the search space between particles. Table 5.4 lists the results for the *gbest* algorithm.

All the *gbest* PSO results are above 0.9, showing that a PSO-based algorithm can be applied to train neural networks as evaluation function for tree-based checkers agents. The best average performance was obtained from small neural networks (only 5 hidden units). A swarm size of 40 particles resulted in the best average performance of 0.9763. Larger swarms cover a wider area of the search space and are therefore more effective in training, even though the time complexity is increased.

The confidence intervals for the experimental results are all relatively small, showing the absence of big fluctuations in the performance.

LBest PSO Approach

This section presents the results for the *lbest* PSO training algorithm. *LBest* PSO uses multiple local best particles instead of a single best particle in the case of *gbest* PSO.

In table 5.5 the results obtained for the *lbest* algorithm do not show a stronger performance than the performance of *gbest* (listed in table 5.4). All the average results for *lbest* PSO are weaker in comparison with *gbest* PSO, with margins of less than 5%.

As in the case of the other algorithms, *lbest* PSO performs the best for smaller neural networks and larger swarms. The best performance of 0.9512 was given by neural networks with 10 hidden units, and a swarm size of 30 particles.

The confidence intervals are also small for *lbest* PSO.

LBest GCPSO Approach

This section presents the results for the *lbest* GCPSO implementation for learning checkers. As in chapter 4, GCPSO algorithm was based on an *lbest* neighborhood structure.

Results for the *lbest* the GCPSO checkers player are listed in table 5.6. All the *lbest* GCPSO scores are relatively close to the maximum 1 score, showing that *lbest* GCPSO is effective in learning the game of checkers.

Interestingly, the results of the *lbest* GCPSO algorithm did not increase significantly with larger swarms. The difference between the average performance of the smallest and larger swarms is just 0.0073 or less than 1%. As with the algorithms above, the best average performance was given by a relatively small neural networks (10 hidden units). Neural networks with 10 hidden units, trained with 30 particles showed the best performance of 0.9668.

Confidence intervals of the results are very small, showing that the results between different games do not vary significantly from one to the other.

5.5.3 Comparison of Results

This section compares the results of the different strategies.

Because of the competition pool used in PSO the relative number of neural networks that are competing are double that of the evolutionary approaches. Refer to section 4.5.3 for a more detailed discussion. The results presented in this section takes this imbalance into account. For a performance comparison where the same number of competitors are used in the competition pool, refer to table 5.7. This table shows the number of individuals and particles used, as well as the relative performance of the relevant algorithms.

Figure 5.3 shows a comparison of different algorithms for different numbers of hidden units used in training. Most of the algorithms perform better for smaller neural networks, i.e, smaller number of hidden units.

Table 5.2: Checkers Performance Results for the Evolutionary Programming Approach with Uniform Mutation

	Hidden units				
Individuals	**5**	**10**	**15**	**20**	**Average**
10	0.9052 ±0.001	0.9224 ±0.0008	0.9140 ±0.0009	0.8970 ±0.0009	*0.9097*
20	0.9586 ±0.0006	0.9515 ±0.0007	0.9425 ±0.0007	0.9343 ±0.0007	*0.9467*
30	0.9587 ±0.0006	0.9571 ±0.0006	0.9545 ±0.0006	0.9514 ±0.0006	*0.9554*
40	0.9661 ±0.0005	0.9614 ±0.0006	0.9546 ±0.0006	0.9498 ±0.0006	*0.9579*
Average	*0.9472*	*0.9488*	*0.9414*	*0.9331*	

The *gbest* variant of PSO is the best performer, except for 10 hidden units. There is approximately 5% difference between the scores of different algorithms as measured by equation (5.4).

Figure 5.4 presents a performance comparison for different algorithms for different swarm and population sizes. Because larger swarm and population sizes cover larger areas of the search space all the algorithms performed better when the swarm or population size was increased. One exception is the *lbest* GCPSO algorithm that only improved marginally with bigger swarms.

The best performing algorithm overall was the *gbest* variation of PSO.

Table 5.3: Checkers Performance Results for Evolutionary Programming Approach with Gaussian Mutation

	Hidden units				
Individuals	**5**	**10**	**15**	**20**	**Average**
10	0.9367 ±0.0013	0.9076 ±0.0016	0.9094 ±0.0015	0.8331 ±0.0022	*0.8967*
20	0.9271 ±0.0014	0.9498 ±0.0011	0.9550 ±0.0011	0.9264 ±0.0014	*0.9396*
30	0.9556 ±0.0011	0.9440 ±0.0012	0.9550 ±0.0011	0.9523 ±0.0011	*0.9517*
40	0.9542 ±0.0011	0.9444 ±0.0012	0.9521 ±0.0012	0.9469 ±0.0012	*0.9494*
Average	*0.9434*	*0.9365*	*0.9429*	*0.9147*	

Table 5.4: Checkers Performance Results for Gbest PSO

	Hidden units				
Individuals	**5**	**10**	**15**	**20**	**Average**
10	0.9548 ±0.0011	0.9329 ±0.0013	0.9194 ±0.0014	0.9378 ±0.0013	*0.9362*
20	0.9485 ±0.0012	0.9471 ±0.0012	0.9450 ±0.0012	0.9480 ±0.0012	*0.9472*
30	0.9763 ±0.0008	0.9719 ±0.0009	0.9510 ±0.0011	0.9337 ±0.0013	*0.9582*
40	0.9609 ±0.0010	0.9633 ±0.0010	0.9551 ±0.0011	0.9584 ±0.0010	*0.9594*
Average	*0.9601*	*0.9538*	*0.9426*	*0.9445*	

Table 5.5: Checkers Performance Results for LBest PSO

	Hidden units				
Individuals	**5**	**10**	**15**	**20**	**Average**
10	0.9227 ±0.0014	0.8448 ±0.0021	0.8843 ±0.0017	0.9104 ±0.0015	*0.8906*
20	0.9463 ±0.0012	0.9340 ±0.0013	0.9152 ±0.0015	0.9268 ±0.0014	*0.9306*
30	0.9476 ±0.0012	0.9314 ±0.0013	0.9501 ±0.0012	0.9296 ±0.0013	*0.9397*
40	0.9512 ±0.0011	0.9409 ±0.0012	0.9301 ±0.0013	0.9302 ±0.0013	*0.9381*
Average	*0.9420*	*0.9128*	*0.9199*	*0.9243*	

Table 5.6: Checkers Performance Results for LBest GCPSO

	Hidden units				
Individuals	**5**	**10**	**15**	**20**	**Average**
10	0.9049 ±0.0016	0.9624 ±0.001	0.9460 ±0.0012	0.9290 ±0.0014	*0.9356*
20	0.9498 ±0.0011	0.9606 ±0.0010	0.9027 ±0.0016	0.9365 ±0.0013	*0.9374*
30	0.9411 ±0.0012	0.9668 ±0.0009	0.9076 ±0.0015	0.9471 ±0.0012	*0.9407*
40	0.9477 ±0.0012	0.9422 ±0.0012	0.9480 ±0.0012	0.9339 ±0.0013	*0.9430*
Average	*0.9359*	*0.9580*	*0.9261*	*0.9366*	

Table 5.7: Checkers Performance Comparison

Situation	1	2
EP Individuals	20	40
PSO Particles	10	20
Evolutionary (Uniform)	0.9467	0.9579
Evolutionary (Gaussian)	0.9396	0.9494
gbest PSO	0.9362	0.9582
lbest PSO	0.8906	0.9397
lbest GCPSO	0.9356	0.9407

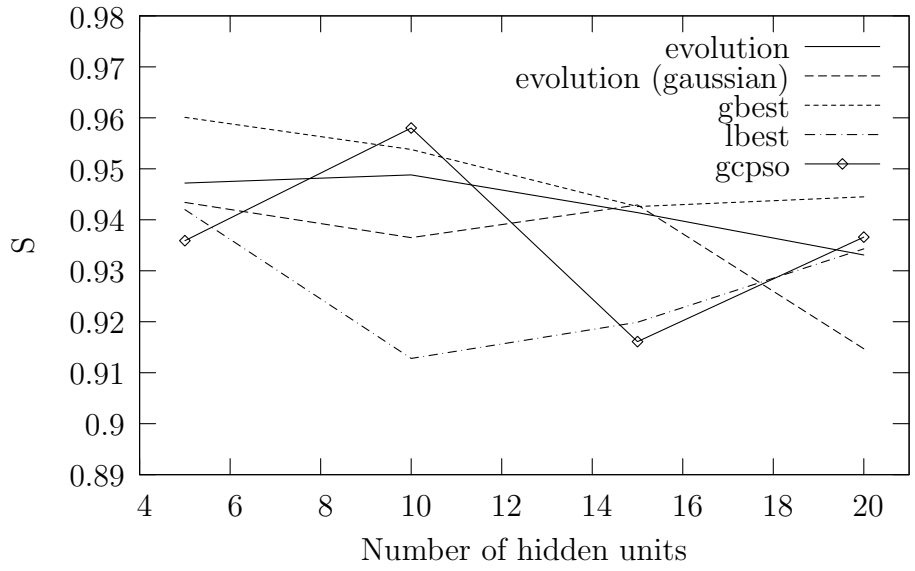

Figure 5.3: Performance Against Hidden Units

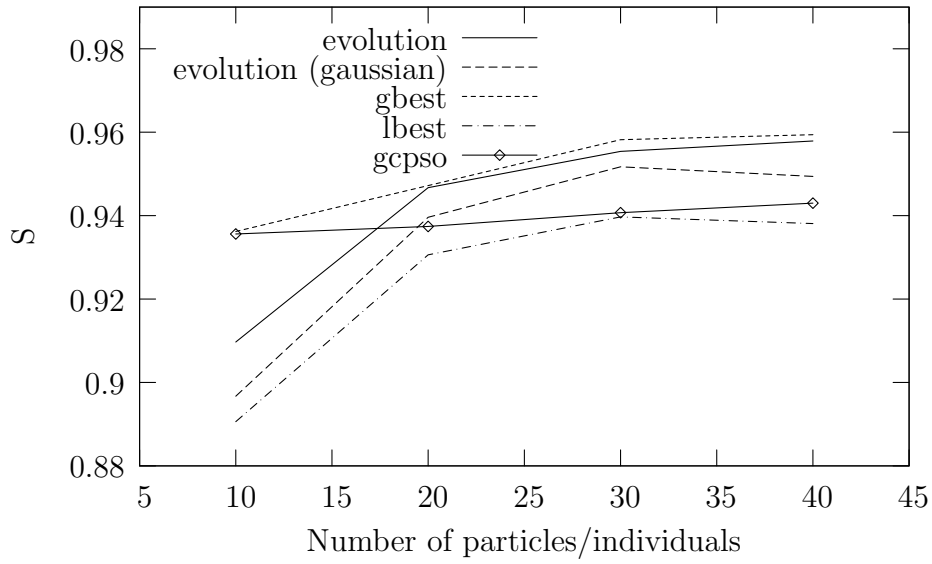

Figure 5.4: Performance Against Swarm/Population Size

5.5.4 Convergence Properties

This section discusses the convergence properties of the training algorithms presented in the previous sections. Convergence in this context is defined as the point where an algorithm is not likely to further improve its solution.

The convergence is calculated as the difference in the mean of the sum of the squares of the weights of the best individual after each iteration. The mean sum of the square weights is given in equation (4.5), while the convergence criterion, $|c(t) - c(t-1)| \rightarrow 0$, is discussed in section 4.5.4.

Figures 5.5 and 5.6 show typical convergence patterns for the evolutionary approach with a uniform mutation. The evolutionary programming approach with uniform mutation did not converge and showed fluctuations throughout the 500 timesteps.

For the evolutionary programming approach with Gaussian mutation the convergence patterns show larger fluctuation in the average values of neural network weights. Although the players tend to become stronger over time, and game play performance is stronger than the

evolutionary programming approach with uniform mutation, these experiments showed an inherent difficulty to converge in the allowed 500 timesteps.

For *gbest* PSO (refer to figure 5.9), a typical pattern for the convergence criterion showed larger fluctuation in the first timesteps and less fluctuations thereafter. Even though less change took place some volatility remained, with none of the experiments converging convincingly in the 500 timesteps.

Figure 5.10 shows a typical convergence pattern for *lbest* PSO. While the algorithm showed larger fluctuations followed by a smoothed pattern in timesteps 0 to 100, the algorithm did move away from the converged point (especially in the last 50 timesteps). *Lbest* experiments showed non-convergent behavior in more extreme cases as illustrated in figure 5.11. These problems appeared in experiments where the swarm size was 10 particles (i.e. the neighborhood size was larger in relation to the swarm size).

Lbest GCPSO was the only approach that showed a typical convergence pattern (refer to figure 5.12), even though experiments with small swarm sizes (10 particles) did not converge (refer to figure 5.13).

5.6 Conclusion

The results presented in this chapter confirmed the results obtained by other research (such as Blondie24), namely that it is possible to train a checkers-playing agent with an evolutionary programming approach, without relying on human knowledge or intervention. It also indicated that it is possible to train a checkers player based on the general framework discussed in chapter 3 with different learning algorithms.

An objective performance measuring function for checkers game agents was introduced. The performance function compares the performance of an agent against that of a random player. A random opponent allowed for a consistent and reproducible measuring criterion. Because of the size of the game tree it is not possible to compute the absolute outcomes of

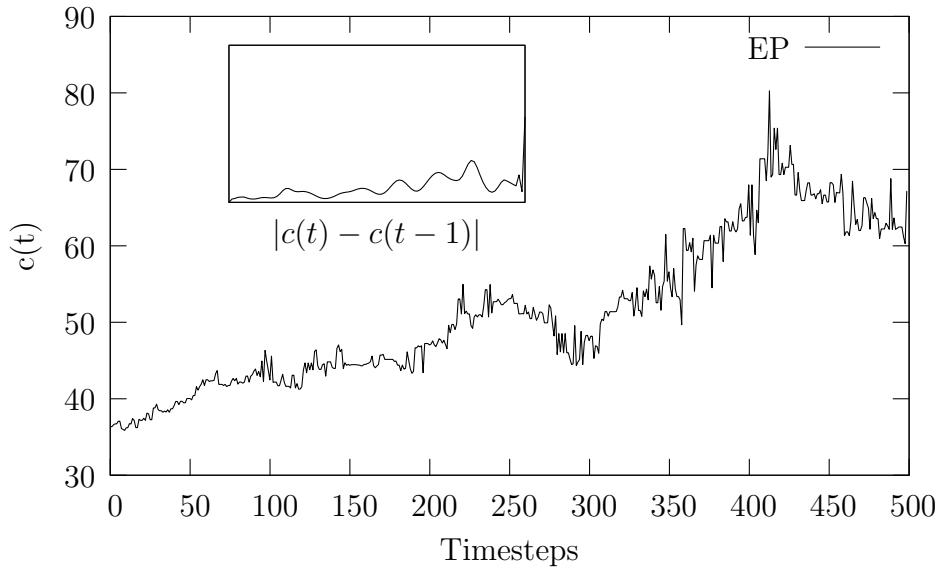

Figure 5.5: Convergence Behavior for the Evolutionary Approach with Uniform Random Mutation (Sample 1)

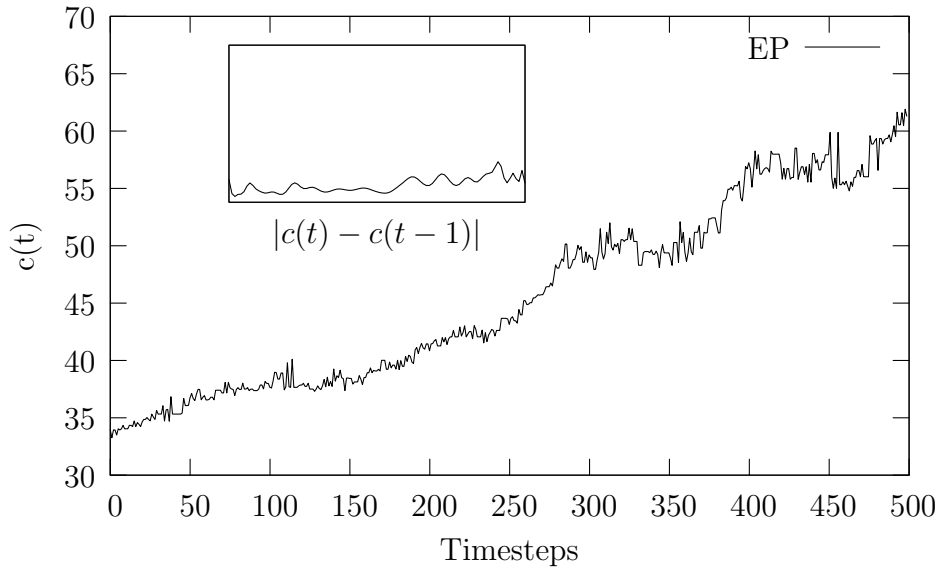

Figure 5.6: Convergence Behavior for the Evolutionary Approach with Uniform Random Mutation (Sample 2)

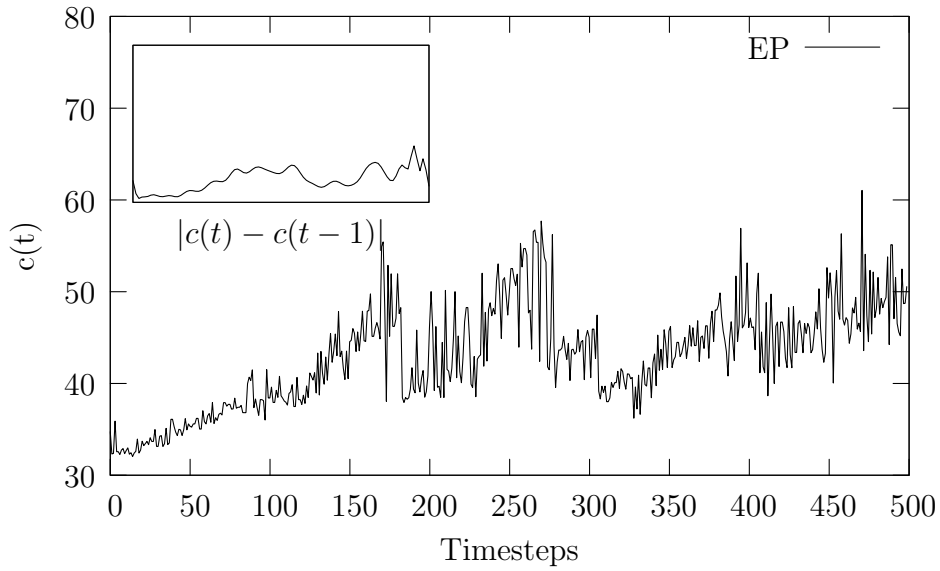

Figure 5.7: Convergence Behavior for the Evolutionary Approach with Gaussian Random Mutation (Sample 1)

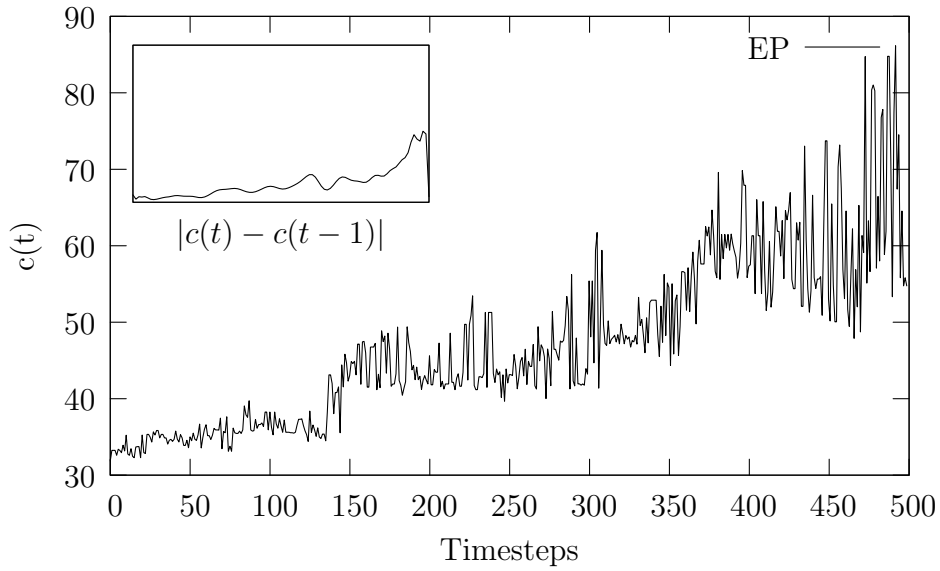

Figure 5.8: Convergence Behavior for the Evolutionary Approach with Gaussian Random Mutation (Sample 2)

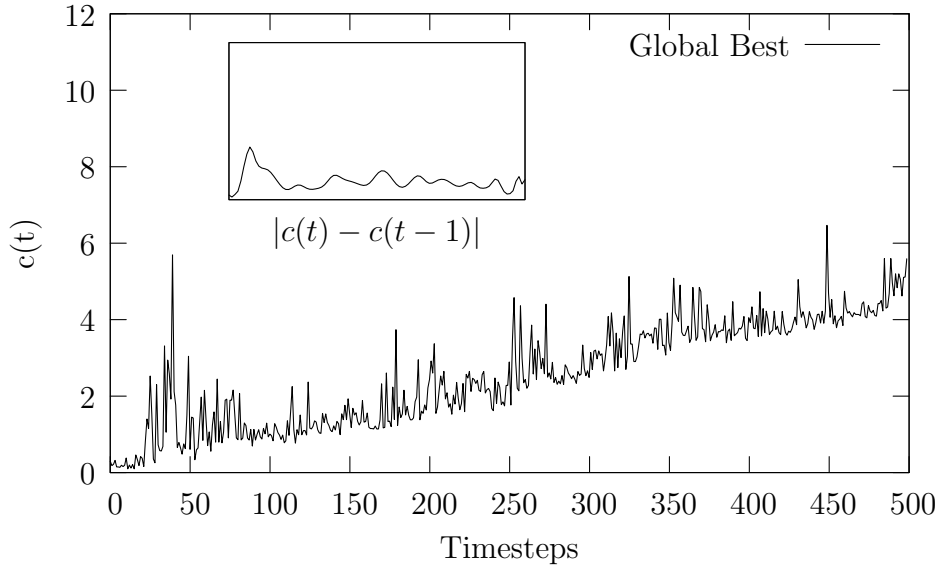

Figure 5.9: Convergence Behavior for *gbest* PSO

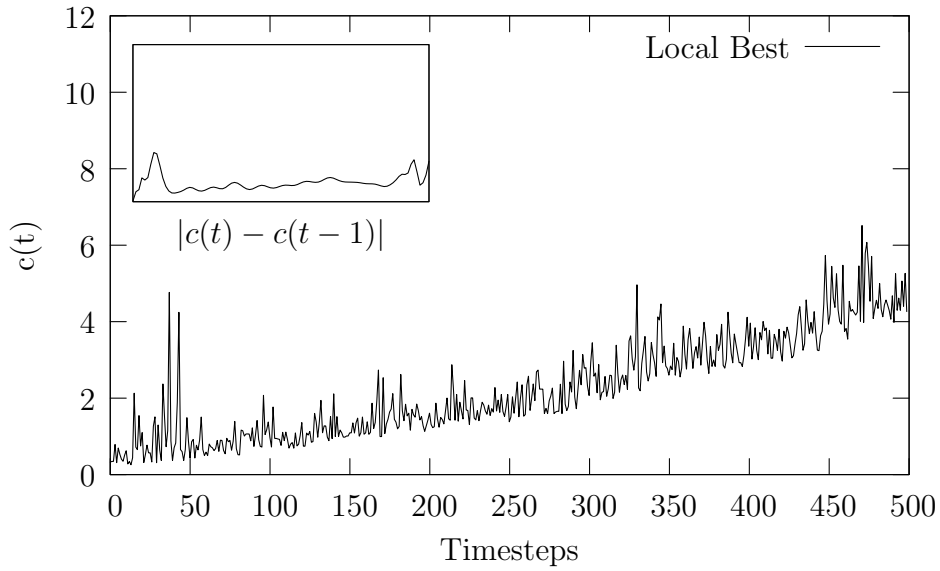

Figure 5.10: Convergence Behavior for *lbest* PSO (Sample 1)

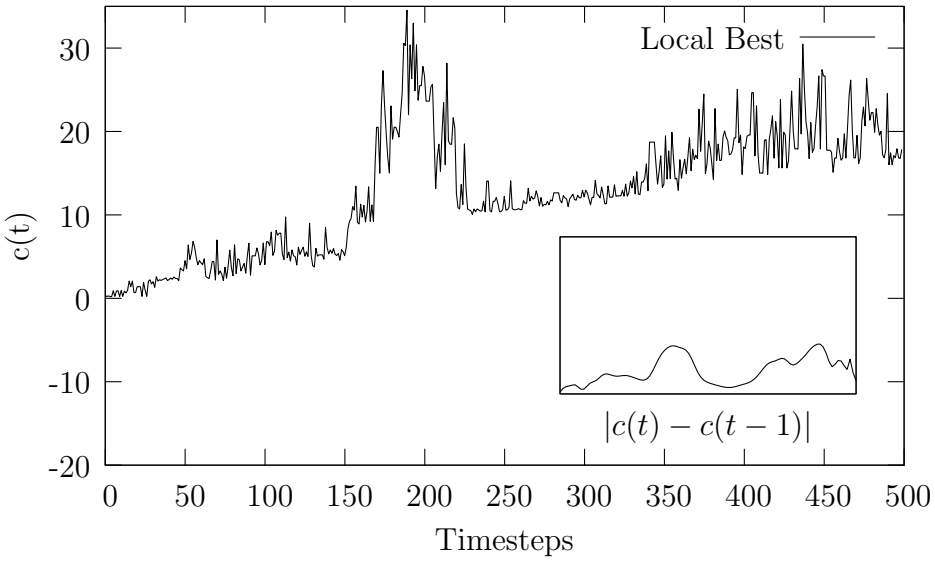

Figure 5.11: Convergence Behavior for *lbest* PSO (Sample 2)

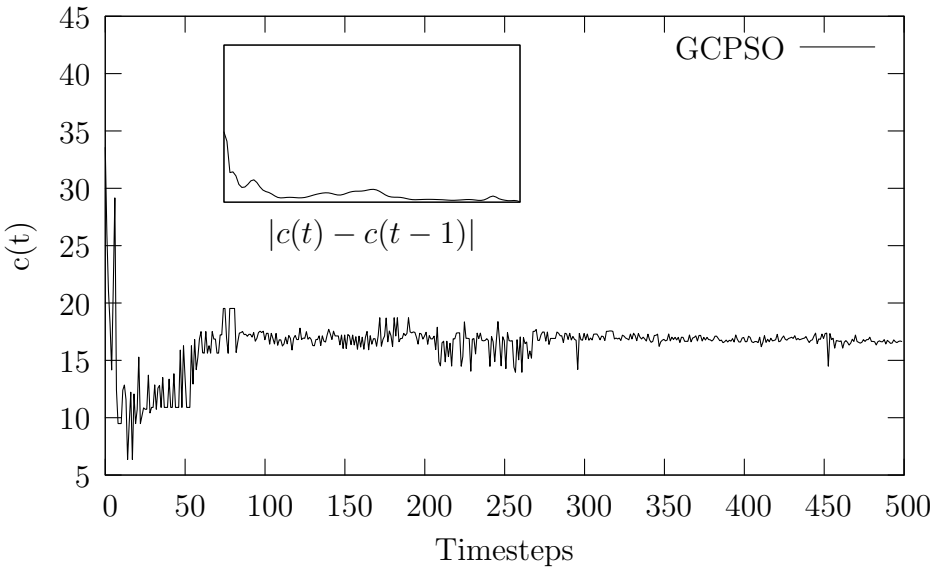

Figure 5.12: Convergence Behavior for *lbest* GCPSO (Sample 1)

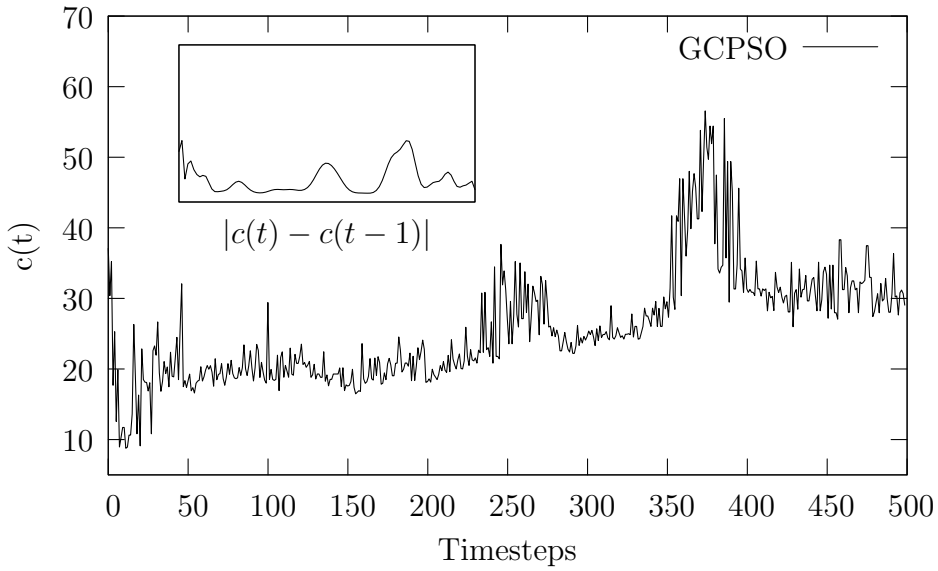

Figure 5.13: Convergence Behavior for *lbest* GCPSO (Sample 2)

the checkers game tree, and a random sample was used to approximate values.

According to the measuring criterion, the evolutionary programming algorithms were able to learn the game of checkers fairly well. PSO algorithms were used as an alternative method to train neural networks to learn checkers. The PSO algorithms were based on the *gbest*, *lbest* and *lbest* GCPSO approaches respectively.

The results listed in this chapter have shown that the PSO algorithm performed on par with the evolutionary programming approach. When comparing the weakest and strongest experiments the *gbest* variation of PSO was the best performer. Although PSO algorithms can overcome local minima problems to a degree, the checkers search space is very complex and remains a challenge, even at 1-ply.

All algorithms performed better for smaller neural networks. The results have shown that smaller neural networks are preferable in training the game of checkers.

In all the evolutionary programming experiments larger population or swarm sizes improved the performance of the checkers playing agents. Larger populations cover a larger part

of the search space and improve performance. The performance of the GCPSO implementation was less sensitive to changes in the swarm size, and experiments with different sized swarms showed similar performance.

Most experiments (with the exception of GCPSO experiments) did not manage to converge successfully.

Chapter 6

Conclusion

This chapter gives a brief summary of the findings and contributions of this thesis. A discussion on future work is also given.

6.1 Summary

This study evaluated evolutionary programming and PSO algorithms as training algorithms of neural network game agents to play the games of tic-tac-toe and checkers.

A general framework for coevolutionary training of neural networks as game-playing agents has been developed and summarized in chapter 3. This model can be used to train game agents for almost any two-player, turn-based, zero sum, perfect knowledge, board game. Different optimization algorithms can be used within the proposed framework. Evolutionary programming and PSO techniques have been applied in this study.

The empirical results confirmed results of other studies [10] that used evolutionary programming to successfully train neural-network-based game agents. This study introduced results for PSO-based algorithms that showed that PSO algorithms can be used as training algorithm to train neural networks to play checkers.

The first objective of this study was to develop an effective measuring criterion for the

play strength of computer game players. The measuring criterion was based on the game agent's performance against a random player (random players pick random moves). The play strength of a random player is strongly correlated with the probabilities of winning, loosing or drawing in the complete game tree. Given enough sample games, a random player provides a consistent play strength. A strong player wins more matches against a random player than predicted by the probabilities, while a weak player beats the random player less often.

The second objective of this study was to apply PSO algorithms to train neural network based tic-tac-toe game agents. Tic-Tac-Toe game playing agents based on evolutionary programming (uniform and Gaussian mutation), *gbest* PSO, *lbest* PSO, and *lbest* GCPSO have been developed. The GCPSO approach was the strongest game agent, as measured against a random tic-tac-toe player. Most algorithms showed convergence for tic-tac-toe training.

The next objective for this study was to develop a PSO training algorithm for neural network checkers agents. As in the case of the tic-tac-toe game agent, algorithms based on evolutionary programming (uniform and Gaussian mutation), *gbest* PSO, *lbest* PSO and *lbest* GCPSO have been tested. Game agents trained with *gbest* PSO had the best performance against a random checkers player at 1-ply. The *gbest* and *lbest* PSO training experiments did generally not converge in the allowed training time, while *lbest* GCPSO converged successfully.

Larger swarm or population sizes performed better than smaller sizes, although GCPSO was less sensitive to different swarm sizes. An increase in neural network size (measured as the number of hidden units) did not improve performance significantly. Game agents that used small numbers of hidden units outperformed game agents with large neural networks.

6.2 Future Work

This section summarizes future work, inspired by this thesis.

6.2.1 Neural Network Optimization Techniques

This study has shown that PSO-trained neural networks can be trained to play board games. The success of this, and other studies [12][94], makes neural network training an avenue to explore.

Future work should be conducted in alternative neural network optimization algorithms. Algorithms that may be considered are:

1. *Alternative PSO implementations.* Many PSO variations have been developed [25] which improve the performance of the basic PSO. Further studies could investigate if these improved algorithms also provides better results for the game learning applications.

2. *Niching algorithms* [68]. Because of the complexity of games, dissimilar strategies may prove successful. Niching methods allow genetic algorithms to maintain a population of diverse individuals. Algorithms that incorporate niching methods are capable of locating multiple, optimal solutions within a single population. PSO niching methods can be applied to game learning applications to evolve multiple play strategies.

3. *Cooperative Swarms [97].* The Cooperative Particle Swarm Optimizer (GCPSO) is a variant of the Particle Swarm Optimizer (PSO) that splits the problem vector, for example a neural network weight vector, across several swarms. Cooperative swarms have been applied to neural network training [98] and may be useful to train neural network game agents.

6.2.2 Neural Network Structure

Standard three-layer, feed-forward neural networks with sigmoid activation functions have been used in this study. Other studies showed that game player performance improves when extra partially connected hidden layers are introduced to include some spatial knowledge of the game [27]. Further research on the effect of different neural network structures should be

undertaken. Different techniques for growing and pruning neural networks should be investigated, to determine optimal neural network structures.

6.2.3 More Complex Games

The techniques reviewed and introduced in this study can be applied to larger, more complex, games such as checkers at n-ply searches, chess, arimaa, and go. These games, especially arimaa and go, pose significant challenges to traditional game-playing algorithms. The game trees of arimaa and go are very large and the evaluation of board positions are problematic. Learning algorithms should be investigated as a means to create successful arimaa and go game agents.

6.2.4 Measuring Performance

The performance of game-playing agents was measured against a random player in this thesis, while other studies used human opponents to measure performance. A random player provides a constant play strength, but human opponents are real-life situations. Future studies can test the correlation of a game agent's performance against a random opponent and human players. Easily repeatable and objective performance measures can be introduced to standardize the evaluation of computational intelligent game agents.

Bibliography

[1] PJ Angeline, JB Pollack, *Competitive Environments. Evolve Better Solutions for Complex Tasks*, Proceedings of the Fifth IEEE International Conference on Genetic Algorithms, 1993.

[2] T Bäck, *Evolutionary Algorithms*, Oxford University Press, 1996.

[3] T Bäck, D Fogel, Z Michalewicz, Eds, *Handbook of Evolutionary Computation*, Oxford University Press, 1997.

[4] CM Bishop, *Neural Networks for Pattern Recognition*, Oxford University Press, 1995.

[5] AD Blair, JB Pollack, *What Makes a Good Co-evolutionary Learning Environment?* Australian Journal of Intelligent Information Processing Systems, Vol 4, pp 166-175, 1997.

[6] B Bouzy, T Cazenave, *Computer Go: An AI Oriented Survey*, Artificial Intelligence, Vol 132, Issue 1, pp 39-103, Elsevier, 2001.

[7] X Cai, DC Wunsch II, *A Parallel Computer-Go Player, Using HDP Method*, Applied Computational Intelligence Laboratory, Deptartment of Electrical and Computer Engineering, University of Missouri, Rolla, USA.

[8] E Cantu-Paz, *A Summary of Research on Parallel Genetic Algorithms*, IlliGAL Technical Report no. 95007, Illinois Genetic Algorithms Laboratory (IlliGAL), Urbana Champaign, IL, USA, July 1995.

[9] EI Chang, RP Lippmann, *Using Genetic Algorithms to Improve Pattern Classification Performance*, Advances in Neural Information Processing Systems, Vol 3, Morgan Kaufmann, pp 797-803, 1991.

[10] K Chellapilla, DB Fogel, *Evolving Neural Networks to Play Checkers Without Expert Knowledge*, IEEE Transactions on Neural Networks, Vol 10, No 6, pp 1382-1391, 1999.

[11] K Chellapilla, DB Fogel, *Evolution, Neural Networks, Games and Intelligence*, Proceedings of the IEEE, Vol 87, No 9, pp 1471-1496, 1999.

[12] K Chellapilla, DB Fogel, *Anaconda Defeats Hoyle 6-0: A Case Study Competing an Evolved Checkers Program Against Commercially Available Software*, Proceedings of the IEEE Congress on Evolutionary Computation, pp 857-863, La Jolla Marriot Hotel, La Jolla, California, USA, July 2000.

[13] Chessbase, *Deep Junior Results*, http://www.chessbase.com/newsdetail.asp?newsid=782, Accessed 2005-03-24.

[14] M Clerc, J Kennedy, *The Particle Swarm Explosion, Stability, and Convergence in a Multidimensional Complex Space*, IEEE Transactions on Evolutionary Computation, Vol 6, No 1, pp 58-73, 2002.

[15] M Cherrity, *A Game Learning Machine*, PhD Thesis, University of California, San Diego, 1993.

[16] C Darwin, *The Origin of Species*, John Murray, 1859.

[17] D DasGupta, Z Michalewicz, *Evolutionary Algorithms in Engineering Applications*, Springer, 2001.

[18] DN Davis, T Chalabi, B Berbank-Green, *Artificial-Life, Agents and GO*, M Mogammadian (ed), New Frontiers in Computational Intelligence and its Applications, Amsterdam, The Netherlands, IOP Press, pp 125-139, 2000.

[19] R Durbin, D Rumelhart, *Product Units: A Computationally Powerful and Biologically Plausible Extension to Backpropagation Networks*, Neural Computation, Vol 1, Issue 1, pp 133-143, 1990.

[20] W Durham, *Co-Evolution: Genes, Culture and Human Diversity*, Stanford University Press, Standford, 1994.

[21] RC Eberhart, X Hu, *Human Tremor Analysis Using Particle Swarm Optimization*. Proceedings of the IEEE Congress on Evolutionary Computation, pp 1927-1930, Washington DC, USA, 1999.

[22] RC Eberhart, J Kennedy, *Swarm Intelligence*, Morgan Kaufmann, 2001.

[23] AE Eiben, T Back, *Empirical Investigation of Multiparent Recombination Operators in Evolution Strategies*, Journal of Evolutionary Computation, No 5, Vol 3, pp 345-365, 1997.

[24] AP Engelbrecht, *Computational Intelligence: An Introduction*, Wiley & Sons, 2002.

[25] AP Engelbrecht, *Fundamentals of Computational Swarm Intelligence*, Wiley & Sons, 2005.

[26] PJ Fleming, RC Purshouse, *Evolutionary Algorithms in Control System Engineering: a Survey*, Control Engineering Practice, Vol 10, pp 1223-1241, 2002.

[27] DB Fogel, *Blondie 24 (Playing At The Edge of AI)*, Morgan Kaufmann Publishers, 2002.

[28] DB Fogel, *Evolutionary Entertainment with Intelligent Agents*, IEEE Computer, Vol 36, Issue 6, pp 106-108, 2003.

[29] DB Fogel, LJ Fogel, VW Porto, *Evolving Neural Networks*, Biological Cybernetics, Vol 63, Issue 6, pp 487-493, 1990.

[30] DB Fogel *Using Evolutionary Programming to Construct Neural Networks that are Capable of Playing Tic-Tac-Toe*, Proceddings of the 1995 IEEE International Conference on Neural Networks, pp 875-880, San Francisco, CA, 1993.

[31] LJ Fogel, AJ Owens, MJ Walsh, *Artificial Intelligence through Simulated Evolution*, Wiley, 1966.

[32] LJ Fogel, *On the organization of intellect*, Doctoral Dissertation, UCLA, 1964.

[33] FD Foresee, MT Hagan, *Gauss-Newton Approximation to Bayesian Regularization*, Proceedings of the International Joint Conference on Neural Networks, 1997.

[34] AS Fraser, *Simulation of Genetic Systems by Automatic Digital Computers*, Australian Journal of Biological Science, Vol 10, pp 484-491, 1957.

[35] AA Freitas, *Data Mining and Knowledge Discovery With Evolutionary Algorithms*, Springer, 2002.

[36] Y Fukuyama, H Yoshida, *A Particle Swarm Optimization for Reactive Power and Voltage Control in Electric Power Systems.*, Proceedings of the IEEE Congress on Evolutionary Computation, Vol 1, pp 87-93, Seoul, Korea, 2001.

[37] SS Ge, CC Hang, TH Lee, T Zhang, *Stable Adaptive Neural Network Control*, Springer, 2001.

[38] S Geman, E Bienenstock, R Doursat, it Neural Networks and the Bias/Variance Dilemma, Neural Computation, No. 4, pp 1-58, 1992.

[39] M Gen, R Cheng, *Genetic Algorithms and Engineering Design* Wiley-Interscience, 1997.

[40] DE Goldberg, K Deb, *A comparative analysis of selection schemes used in genetic algorithms*, Foundations of Genetic Algorithms, GJE Rawlins, edt., pp 69-93, 1991.

[41] DE Goldberg, J Richardson, *Genetic Algorithm With Sharing for Multimodal Function Optimization*, Proceedings of the Second IEEE International Conference on Genetic Algorithms, Vol 1, pp 41-49, 1987.

[42] P Grosso, *Computer Simulations of Genetic Adaptation: Parallel Subcomponent Interaction in a Multilocus model*, PhD thesis, University of Michigan, 1985.

[43] K Gurney, *An Introduction to Neural Networks*, Routledge, 1997.

[44] J Hertz, RG Palmer, AS Krogh, *Introduction to the Theory of Neural Computation*, Perseus Books, 1990.

[45] JH Holland, *ECHO: Explorations of Evolution in a Minature World*, Proceedings of the Second IEEE Conference on Artificial Life, Redwood City, CA, Addison-Wesley 1990.

[46] K Hornik, M Stinchcombe, H White, *Multilayer Feedforward Networks are Universal Approximators*, Neural Networks, Vol 2, Issue 5, pp 359-366, 1989.

[47] IBM Research, *Deep Blue*, http://www.research.ibm.com/deepblue, Accessed 2002-09-24.

[48] A Ismail, AP Engelbrecht, *Training Product Units in Feedforward Neural Networks using Particle Swarm Optimization*, Proceedings of the IEEE International Conference on Artificial Intelligence, Durban, South Africa, 1999.

[49] W Jakob, M Gorges-Schleuter, C Blume, *Application of Genetic Algorithms to Task Planning and Learning*, Manner and Manderick, editors, Parallel Problem Solving from Nature PPSN'2, LNCS, pp 291-300, 1992.

[50] T Jones, *Crossover, Macromutation, and Population-based Search*, Proceedings of the Sixth International Conference on Genetic Algorithms, San Francisco, CA, pp 73-80, 1995.

[51] J Kennedy, *Stereotyping: Improving Particle Swarm Performance With Cluster Analysis*, Proceedings of the IEEE Congress on Evolutionary Computing, Vol 2, pp 303-308, San Diego, USA, 2000.

[52] J Kennedy, *Small Worlds and Mega-Minds: Effects of Meighborhood Topology on Particle Swarm Performance*, Proceedings of the IEEE Congress on Evolutionary Computation, Vol 3, pp 1931-1938, Washington DC, USA, 1999.

[53] J Kennedy, RC Eberhart, *Particle Swarm Optimization*, Proceedings of the IEEE International Joint Conference on Neural Networks, Vol 4, pp 1942-1948, Perth, Australia, 1995.

[54] J Kennedy, R Mendes, *Population structure and Particle Swarm Peformance*, Proceedings of the IEEE Congress on Evolutionary Computing, Honululu, Hawai, 2002.

[55] T Kohohen, *Self-Organizing Maps*, Third Extended Edition, Heidelberg, New York, 2001.

[56] T Kohohen, *Improved Versions of Learning Vector Quantization*, Proceedings of the IEEE International Joint Conference on Neural Networks, Vol 1, San Diego, pp 545-550, 1990.

[57] RE Korf, *Depth-first Iterative-Deepening: An Optimal Admissible Tree Search*, Artificial Intelligence 27, Vol 1, pp 97-109, 1985

[58] JR Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.

[59] S Hochreiter, Y Bengio, P Frasconi, J Schmidhuber, *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies*, SC Kremer, JF Kolen, editors, A Field Guide to Dynamical Recurrent Neural Networks, IEEE Press, 2001.

[60] R Lange, R Muller, *Quantifying a Critical Training Set Size For Generalization and Overfitting Using Teacher Neural Networks*, Proceedings of the IEEE International Conference on Artifical Neural Networks, Vol 1, pp 475-500, London, 1994.

[61] J Larsen, LK Hansen, *Generalization Performance of Regularized Neural Network Models*, Proceddings of the 4th IEEE Workshop on Neural Networks for Signal Processing, Ermioni, Greece, pp 42-51, 1994.

[62] DNL Levy, *How Computers Play Chess*, W H Freeman & Co (Sd), 1991.

[63] SJ Louis, Z Xu, *Genetic Algorithms for Open Shop Scheduling and Rescheduling*, ME Cohen, DL Hudson, editors, ISCA Eleventh International Conference on Computers and their Applications, pp 99-102, 1996.

[64] J Love, J Hodgkins, *Chess Ideas For Young Players*, G Bell and Sons (Ltd), London, 1962.

[65] A Lubberts, R Miikkulainen, *Co-Evolving a Go-Playing Neural Network*, Proceedings of the IEEE Genetic and Evolutionary Computation Conference, pp 14-19, San Francisco, 2001.

[66] GF Luger, WA Stubblefield, *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, Addison-Wesley, Third Edition, 1997.

[67] M MacFadyen, *The Game of Go*, Carlton, 1998.

[68] SW Mahfoud, *Niching Methods for Genetic Algorithms*, Ph.D. thesis, Department of General Engineering, IlliGAL Report 95001, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1995.

[69] B Manderick, P Spiessens, *Fine-Grained Parallel Genetic Algorithms*, Proceedings of the Third IEEE International Conference on Genetic Algorithms, Fairfax, USA, 1989.

[70] T Masters, *Practical Neural Network Recipes in C++*, Morgan Kaufmann, 1993.

[71] L Messerschmidt, AP Engelbrecht, *Learning to Play Games Using a PSO-Based Competitive Learning Approach*, IEEE Transactions on Evolutionary Computation, Vol 8, No 3, pp 280-288, 2004.

[72] TM Mitchell, *Machine Learning*, McGraw-Hill, Portland, Oregon, USA, 1997.

[73] NJ Nilsson, *Artificial Intelligence: A new Sythesis*, Morgan Kaufmann Publishers, 1998.

[74] R Pask, *Starting out In Checkers*, Everyman Chess, 2001.

[75] T Peram, K Veeramachaneni, CK Mohan, *Fitness-Distance-Ratio Based Particle Swarm Optimization* IEEE Swarm Intelligence Symposium, Indiana, USA, pp 174-181, 2003.

[76] V Petridis, VG Kaburlasos, *Fuzzy Lattice Neural Network (FLNN):A Hybrid Model for Learning*, IEEE Transactions on Neural Networks, Vol 9, No 5, pp 877-890, 1998.

[77] MA Potter, *The Design and Analysis of a Computational Model of Cooperative Coevolution* Ph.D. Thesis, George Mason University, 1997.

[78] MA Richards, GA Shaw, *Chips, Architectures and Algorithms: Reflections on the Exponential Growth of Digital Signal Processing Capability*, IEEE Signal Processing Magazine, 2004.

[79] BD Ripley, *Pattern Recognition and Neural Networks*, Cambridge University Press, 1996.

[80] CD Rosin, RK Belew, *Finding Opponents Worth Beating: Methods for Competative Co-Evolution*, Proceedings of the Sixth IEEE International Conference on Genetic Algorithms, 1995.

[81] H Rowley, S Baluja, T Kanade, *Human Face Detection in Visual Scenes*, Advances in Neural Network Information Processing Systems, Vol 8, pp 875-881, 1996.

[82] A Sakar, RJ Mammone, *Growing and Pruning Neural Tree Networks*, IEEE Transactions on Computers, Vol 42, No 3, pp 291-299, 1993.

[83] J Salerno, *Using The Particle Swarm Optimization Technique to Train a Recurrent Neural Model*, In Proceedings of the Ninth IEEE International Conference On Tools With Artificial Intelligence, pp 45 - 49, 1997.

[84] J Schaeffer, R Lake, P Lu and M Bryant, *Chinook: The Man-Machine World Checkers Champion*, AI Magazine, Vol 17, No 1, pp 21-29, 1996.

[85] TJ Sejnowski, CR Rosenberg, *Parallel Networks that Learn to Pronounce English Text*, Complex Systems, Vol 1, No 1, pp 145-168, 1987.

[86] CE Shannon, *A Chess Playing Machine*, Scientific American, 1950.

[87] C Shannon, *Programming a Computer for Playing Chess*, Philosophical Magazine, Vol 41, No 314, pp 256-275, 1950.

[88] Y Shi, RC Eberhart, *An Empirical Study of Particle Swarm Optimization*, Proceedings of the IEEE Congress on Evolutionary Computation, Washington DC, Vol 1, USA, pp 1945-1949, 1999.

[89] Y Shi, RC Eberhart, *A modified particle swarm optimizer* Proceedings of the IEEE World Conference on Computational Intelligence, pp 69-73, Anchorage, Alaska, 1998.

[90] H Simon, A Newell, *Heuristic Problem Solving: The Next Advance in Operations Research*, Operations Research, Vol 6, pp 1-10, 1958.

[91] H Simon, J Schaeffer, *The Game of Chess*, Handbook of Game Theory with Economic applications, Aumann, Hart, editors, North Holland, 1992.

[92] O Syed, *Arimaa*, http://www.arimaa.com, Accessed 2005-04-01.

[93] JA Snyman, *An Improved Version of the Original LeapFrog Dynamic Method for Unconstrained Minimization*, Appl Math Modelling, Vol 7, pp 216-218 1983.

[94] G Tesauro, *Temporal Difference Learning and TD-backgammon*, Communications of the ACM, Vol 38, No 3, pp 58-68, 1995.

[95] S Thrun, *Learning To Play the Game of Chess*, Advances in Neural Information Processing Systems, Vol 7, G Tesauro, D Touretzky, T Leen, editors, MIT Press, 1995.

[96] F van den Bergh, *An Analysis of Particle Swarm Optimizers*, PhD Thesis, Department of Computer Science, University of Pretoria, South Africa, 2002.

[97] F van den Bergh, AP Engelbrecht, *Cooperative Learning in Neural Networks Using Particle Swarm Optimizers*, South African Computer Journal, Vol 26, pp 84 - 90 2000.

[98] F van den Bergh, AP Engelbrecht, *Using Cooperative Particle Swarm Optimization to Train Product Unit Neural Networks*, IEEE International Joint Conference on Neural Networks, Washington DC, USA, 2001.

[99] A Waibel, T Hanazawa, G Hinton, *Phoneme Recognition Using Time-Delay Neural Networks*, IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol 37, No 3, pp 329-339, 1989.

[100] E Wan, *Neural network classication: a Bayesian interpretation*, IEEE Transactions on Neural Networks 1, Vol 4, pp 303-305, 1990.

[101] A Weigend, *On Overfitting and the Effective Number of Hidden Units*, Proceedings of the 1993 Connectionist Models Summer School, pp 335-342 1993.

[102] PJ Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, PhD thesis, Department of Applied Mathematics, Harvard University, 1974.

[103] X Yao, *Evolving Artificial Neural Networks*, Proceedings of the IEEE, Vol 89, No 9, pp 1423-1447, 1999.

Appendix A

Derived Publications

This appendix lists all publications derived from this study:

1. L Messerschmidt, AP Engelbrecht,
   Learning to play games using a pso-based competitive learning approach, Proceedings of the 4th Asia-Pacific Conference on Simulated Evolution and Learning, Singapore, 2002.

2. L Messerschmidt, AP Engelbrecht,
   Learning to Play Games Using a PSO-Based Competitive Learning Approach IEEE Transactions on Evolutionary Computation, Volume 8, Issue 3, p. 280 - 288, 2004.