Evolution of Cartesian Genetic Programs Capable of Learning

Gul Muhammad Khan
Electrical Engineering Department
NWFP UET Peshawar
Pakistan
gk502@nwfpuet.edu.pk

Julian F. Miller
Intelligent System Design Group
Electronics Department
University of York
jfm7@ohm.york.ac.uk

ABSTRACT

We propose a new form of Cartesian Genetic Programming (CGP) that develops into a computational network capable of learning. The developed network architecture is inspired by the brain. When the genetically encoded programs are run, a networks develops consisting of neurons, dendrites, axons, and synapses which can grow, change or die. We have tested this approach on the task of learning how to play checkers. The novelty of the research lies mainly in two aspects: Firstly, chromosomes are evolved that encode programs rather than the network directly and when these programs are executed they build networks which appear to be capable of learning and improving their performance over time solely through interaction with the environment. Secondly, we show that we can obtain learning programs much quicker through co-evolution in comparison to the evolution of agents against a minimax based checkers program. Also, co-evolved agents show significantly increased learning capabilities compared to those that were evolved to play against a minimax-based opponent.

Categories and Subject Descriptors

I.2.2 [**ARTIFICIAL INTELLIGENCE**]: Automatic Programming—*Program synthesis*; I.2.6 [**ARTIFICIAL INTELLIGENCE**]: Learning—*Connectionism and neural nets*

General Terms

Algorithms, Design, Performance

Keywords

Cartesian Genetic Programming, Computational Development, Co-evolution, Artificial Neural Networks, Checkers

1. INTRODUCTION

In our view the process of biological development underpins learning. Since in biology all learning occurs during development, and DNA does not in itself encode learned information. This raises the question: How is a *capability* for learning encoded at a genetic level? We are also interested in finding out how important for learning, is the interaction between two systems developing in response to each other? In this paper, we evolve genotypes that encode programs that *when executed* gives rise to a neural network that plays checkers. In particular, we demonstrate how important it is to co-evolve and co-develop two agents, instead of evolving and developing a single agent for learning.

Following Khan et al. the genotype we evolve is a set of computational functions that are inspired by various aspects of biological neurons [10]. Each agent (player) has a genotype that grows a computational neural structure (phenotype). The initial genotype that gives rise to the dynamic neural structure is obtained through evolution. As the number of evolutionary generations increases the genotypes develop structure that allow the players to play checkers increasingly well.

Our method employs very few, if any, of the traditional notions that are used in the field of Artificial Neural Networks. Unlike traditional ANNs we do not evolve or directly adjust set of numbers that defines a network. We run evolved programs that can adjust the network indefinitely. This allows our network to learn while it develops during its lifetime. The network begins as small randomly defined networks of neurons with dendrites and axosynapses. The job of evolution is to come up with genotypes that encode *programs* that when *executed* develop into mature neural structures that learn through environmental interaction and continued development.

ANNs can only solve a specific problem as they model learning through synaptic weights. Whereas memory and learning in brains is caused by many other mechanisms. Synaptic weights are only responsible for extremely short term memory. Also if very complex tasks are required to be solved with say, billions of weights, current traditional approaches won't scale. In principle ours will as the network complexity is not related to the complexity of the evolved programs. So in a nutshell we choose to model at this particular level of abstraction because we feel it has the plasticity we need and will scale better. What we do is *inspired* by biology. We are not trying to model biology. We expect the additional model complexity to pay off when we allow it to develop in interaction with the environment over long time scales and on different problems simultaneously.

There are a number of techniques in which an agent can

GECCO'09, July 8–12, 2009, Montréal Québec, Canada.

be trained to learn. One is to train the agent against a well-trained agent whose capabilities do not change at runtime, i.e evolve and develop the system against a minimax-based checkers software program. The second method is to co-evolve two agents that have almost equal learning capabilities, so their level of play improves in response to each other.

In the first case agents develop during the course of a series of games playing against a fixed level minimax program that plays checkers. At the end of the evolutionary run we test the well evolved agent against a series of lesser evolved ancestor agents to see how the learning capabilities have increased over evolutionary time. To do this we allowed well evolved agents to play a number of games against its opponent from various generation, and checked if the level of play is improved. From the results shown later it is clearly evident that the learning capability of the agents improves over evolutionary time. In second case we have co-evolved agents against each other allowing both agents to develop over five game series. We have evolved them for one thousand (1000) generations and then tested the highly evolved agents against less evolved agents. Once again our results show that on average the highly evolved agents perform much better than the lesser evolved agents.

In order to test whether co-evolution or evolution produced better learning agents, we have tested the agents from various generation of co-evolution and evolution. From the results it is evident that co-evolution improves the learning capability of the agents much more than that of the evolved agents.

2. CARTESIAN GENETIC PROGRAMMING (CGP)

CGP is a well established and effective form of Genetic Programming. It represents programs by directed acyclic graphs [13]. The genotype is a fixed length list of integers, which encode the function of nodes and the connections of a directed graph. Nodes can take their inputs from either the output of any previous node or from a program input (terminal). The phenotype is obtained by following the connected nodes from the program outputs to the inputs. For our checkers work, we have used function nodes that are variants of binary if-statements known as 2 to 1 multiplexers [12] as shown in figure 1.

The four functions in figure 1 are the possible input combinations of a three input (two inputs and a control) multiplexer, when inputs are either inverted or not. Multiplexers can be considered as atomic in nature as they can be used to represent any logic function [12].

Figure 1 shows the genotype and the corresponding phenotype obtained connecting the nodes as specified in the genotype. The Figure also shows the inputs and outputs to the CGP. Output is taken from the nodes as specified in the genotype (6, 8, 4). In our case we have not specified the output in the genotype and have used a fixed pseudo random list of numbers to specify where the output should be taken from.

In CGP an evolutionary strategy of the form $1 + \lambda$, with λ set to 4 is often used [12]. The parent, or elite, is preserved unaltered, whilst the offspring are generated by mutation of the parent. If two or more chromosomes achieve the highest fitness then *newest* (genetically) is always chosen.

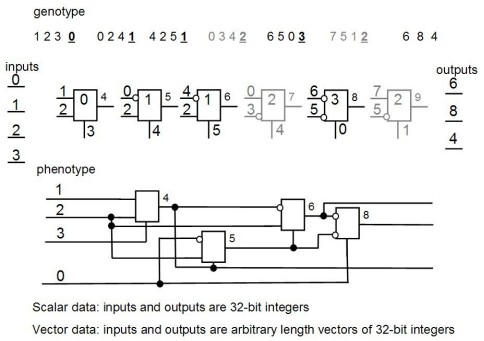

Figure 1: Structure of CGP chromosome. Showing a genotype for a 4 input, 3 output function and its decoded phenotype. Inputs and outputs can be either simple integers or an array of integers. Note nodes and genes in grey are unused and small open circles on inputs indicate inversion. The function gene in genotype is underlined. All the inputs and outputs of multiplexers are labeled. Labels on the inputs of the multiplexer shows where are they connected (i.e. they are addresses). Input to CGP is applied through the input lines as shown in figure. The number of inputs (four in this case) and outputs (three in this case) to the CGP is defined by the user, which is different from the number of inputs per node (three in this case i.e. a, b and c.)

3. CO-EVOLUTIONARY COMPUTATION

Co-evolutionary algorithms are generally used for artificial life, optimization, game learning and machine learning problems. Co-evolutionary computation is largely used in the competitive environment. These interactions can be either between individuals competing in a game context [19, 21] or between different populations competing in predator/prey type relationships [7, 17, 21].

In competitive co-evolution an individual's fitness is evaluated based on its performance against the opponent in the population. Fitness shows the relative strengths of solutions not the absolute solutions, thus causing the opponent fitness to decrease relatively. These competing solutions will create an "Arms Race" of increasingly better solutions [3, 21]. The feedback mechanisms between individuals based on their selection produces a strong force toward increased complexity [18].

Nolfi and Floreano co-evolved two competing populations of predator and prey robots in order to emphasize how lifetime learning allows evolving individuals to achieve generality, i.e. the ability to produce effective behavior in a variety of different circumstances [15]. What is interesting about this experimental situation is that, since both populations change across generations, predators and prey face ever-changing and potentially progressively more complex challenges. They also observed that, in this situation, evolution alone displays severe limitations and progressively more solutions can be developed only by allowing evolving individuals to adapt on the fly through a form of lifetime learning.

In recent years, co-evolutionary techniques have been applied to several games, including Othello [14], Go [11], Chess [9], and Checkers [24] [5].

4. RELATED DEVELOPMENT MODELS

Nolfi et al presented a model in which the genotype-phenotype mapping (i.e. ontogeny) takes place during the individual's lifetime and is influenced both by the genotype and by the external environment [16]. The 2D neural networks adapt during their lifetime to different environments. The neurons had no dendrites only upward growing axons. Connections between neurons happens when axons arrive in the vicinity of another neuron. The activity of a neuron affects whether its axon grows or not, in this way they linked lifetime 'electrical' activity with morphological structure

Cangelosi proposed a related neural development model, which starts with a single cell undergoing a process of cell division and migration [1]. This continues until a collection of neurons arranged in 2D space is developed. These neurons grow their axons to produce connection among each other to develop a neural network. The rules for cell division and migration is specified in genotype, for a related approach see [2, 6].

Rust and Adams devised a developmental model coupled with a genetic algorithm to evolve *parameters* that grow into artificial neurons with biologically-realistic morphologies [22]. They also investigated activity dependent mechanisms [23] so that neural activity would influence growing morphologies.

Jakobi created an artificial genomic regulatory network [8]. He used proteins to define neurons with excitatory or inhibitory dendrites. The individual cell divides and moves due to protein interactions with an artificial genome, causing a complete multicellular network to develop. After differentiation each cell grows dendrites following chemical sensitive growth cones to form connections between cells. This develops a complete conventional recurrent ANN, which is used to control a simulated Khepera robot for obstacle avoidance and corridor following.

Federici presented an indirect encoding scheme for development of a neuro-controller [4]. The adaptive rules used were based on the correlation between post-synaptic electric activity and the local concentration of synaptic activity and refractory chemicals. Federici used two steps to produced the neuro-controllers: A growth program (implemented as a simple recurrent neural network) in a genotype to develop the whole multi-cellular network in the form of a phenotype and a translation step where cells are interpreted as spiking neurons.

Roggen et al. devised a hardware cellular model of developmental spiking ANNs [20]. Each cell can hold one of two types of fixed input weight neurons, excitatory or inhibitory each with one of 5 fixed possible connection arrangements to neighbouring neurons. In addition each neuron has a fixed weight external connection. The neuron integrates the weighted input signals and when it exceeds a certain membrane threshold it fires. This is followed by a short refractory period. They have a leakage which decrements membrane potentials over time.

In almost all previous work the internal functions of neurons were either fixed or only parameters were evolved. Connections between neurons are simple wires instead of complicated synaptic process. Development stops once the evaluation is started so there is no development in real time. The model we propose is inspired by the characteristics of real neurons.

5. KEY FEATURES AND BIOLOGICAL BASIS FOR THE MODEL

Features of biological neural systems that we think are important to include in our model(Cartesian Genetic Programming Computational Network (CGPCN)) are synaptic transmission, and synaptic and developmental plasticity. Signalling between biological neurons happens largely through synaptic transmission, where an action potential in the pre-synaptic neuron triggers a short lasting response in the post-synaptic neuron [25]. In our model signals received by a neuron through its dendrites are processed and a decision is taken whether to fire an action potential or not. Table 1 lists all the properties of biological systems that are incorporated into our model. Table 1 also shows the presence and absence of these properties in existing ANNs and neural development models.

Neurons in biological systems are in constant state of change, their internal processes and morphology change all the time based on the environmental signals. The development process of the brain is strongly affected by external environmental signals. This phenomenon is called Developmental Plasticity. Developmental plasticity usually occurs in the form of synaptic pruning [26]. This process eliminates weaker synaptic contacts, but preserves and strengthens stronger connections. More common experiences, which generate similar sensory inputs, determine which connections to keep and which to prune. More frequently activated connections are preserved. Neuronal death occurs through the process of apoptosis, in which inactive neurons become damaged and die. This plasticity enables the brain to adapt to its environment.

A form of developmental plasticity is incorporated in our model, branches can be pruned, and new branches can be formed. This process is under the control of a 'life cycle' chromosome (described in detail in section 6) which determines whether new branches should be produced or branches need to be pruned. Every time a branch is active, a life cycle program is run to establish whether the branch should be removed or should continue to take part in processing, or whether a new daughter branch should be introduced into the network.

Starting from a randomly connected network, we allow branches to navigate (Move from one grid square to other, make new connections) in the environment, according to the evolutionary rules. An initial random connectivity pattern is used to avoid evolution spending extra time in finding connections in the early phase of neural development.

Changes in the dendrite branch weight are analogous to the amplifications of a signal along the dendrite branch, whereas changes in the axon branch (or axo-synaptic) weight are analogous to changes at the pre-synaptic level and post-synaptic level (at synapse). Inclusion of a soma weight is justified by the observation that a fixed stimulus generates different responses in different neurones.

Through the introduction of a 'life cycle' chromosome, we have also incorporated developmental plasticity in our model. The branches can self-prune and can produce new branches to evolve an optimized network that depends on the complexity of the problem [26].

Name	ANNs	Neural development	Biology	CGPCN
Neuron Structure	Node with connections	Node with axons and dendrites	Soma with dendrites, axon and dendrite branches	Soma with dendrites, axon and dendrite branches
Interaction of branches	No	No	Yes	Yes
Neural function	Yes	Yes	Yes	Yes
Resistance	No	Yes/No	Yes	Yes
Health	No	No	Yes	Yes
Neural Activity	No	No	Yes	Yes
Synaptic Communication	No	No	Yes	Yes
Arrangement of Neurons	Fixed	Fixed	Arranged in space (Dynamic Morphology)	Arranged in Artificial space (Dynamic Morphology)
Spiking (Information processing)	Yes, but not all	Yes, but not all	Yes	Yes
Synaptic Plasticity	Yes	No	Yes	Yes
Developmental Plasticity	Yes	No	Yes	Yes
Arbitrary I/O	No	No	Yes	Yes
Learning Rule	Specified	Specified	Unspecified	Unspecified
Activity Dependent Morphology	No	Some	Yes	Yes

Table 1: List of all the properties of biological systems that are incorporated into CGPCN or are present in ANNs and neural development models.

6. THE CGP COMPUTATIONAL NETWORK (CGPCN)

This section describes in detail the structure of the CGPCN, along with the rules and evolutionary strategy used to run the system.

In the CGPCN neurons are placed randomly in a two dimensional spatial grid so that they are only aware of their spatial neighbours (as shown in figure 2). Each neuron is initially allocated a random number of dendrites, dendrite branches, one axon and a random number of axon branches. Neurons receive information through dendrite branches, and transfer information through axon branches to neighbouring neurons. The dynamics of the network also changes, since branches may grow or shrink and move from one CGPCN grid point to another. They can produce new branches and can disappear, and neurons may die or produce new neurons. Axon branches transfer information only to dendrite branches in their proximity. Electrical potential is used for internal processing of neurons and communication between neurons, and we represent it as an integer.

Health, Resistance, Weight and Statefactor

Four integer variables are incorporated into the CGPCN, representing either fundamental properties of the neurons (*health*, *resistance*, *weight*) or as an aid to computational efficiency (*statefactor*). The values of these variables are adjusted by the CGP programs. The *health* variable is used to govern replication and/or death of dendrites and connections. The *resistance* variable controls growth and/or shrinkage of dendrites and axons. The *weight* is used in calculating the potentials in the network. Each soma has only two variables: *health* and *weight*. The *statefactor* is used as a parameter to reduce computational burden, by keeping some of the neurons and branches inactive for a number of cycles. Only when the *statefactor* is zero are the neurons and branches considered to be active and their corresponding program is run. The value of the *statefactor* is affected indirectly by CGP programs. The bio-inspiration for the *statefactor* is the fact that not all neurons and/or dendrites branches in the brain are actively involved in each process.

6.1 Inputs, Outputs and Information Processing in the Network

The external inputs (encoding a simulated potential) are applied to the CGPCN and presented to axo-synaptic electrical processing chromosomal branches as shown in figure 3. These are distributed in the network in a similar way to the axon branches of neurons. After this the program encoded in the axo-synaptic electrical branch chromosome is executed, and the resulting signal is transferred to its neighbouring active dendrite branches. Similarly we have outputs which read the signal from the CGPCN through dendrite branches. These branches are updated by the axo-synaptic chromosomes of neurons in the same way as other dendrite branches and after five cycles the potentials produced are averaged and this value is used as the external output.

Information processing in the network starts by selecting the list of active neurons in the network and processing them in a random sequence. Each neuron take the signal from the dendrites by running the electrical processing in dendrites. The signals from dendrites are averaged and applied to the soma program along with the soma potential. The soma program is run to get the final value of soma potential, which decides whether a neuron should fire an action potential or not. If the soma fires, an action potential is transferred

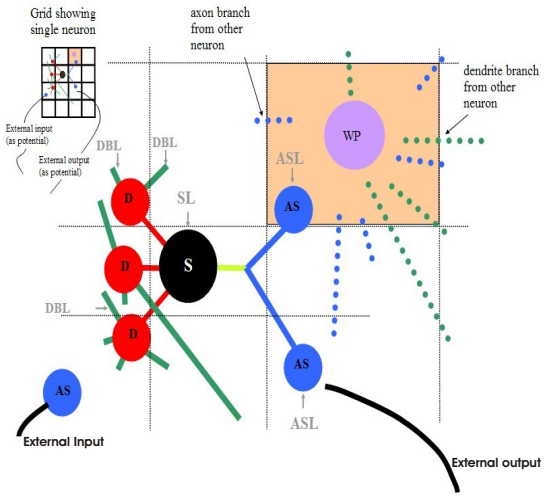

Figure 2: On the top left a grid is shown containing a single neuron. The rest of the figure is an exploded view of the neuron is given. The neuron consists of seven evolved computational functions. Three are 'electrical' and process a simulated potential in the dendrite (D), soma (S) and axo-synapse branch (AS). Three more are developmental in nature and are responsible for the 'life-cycle' of neural components (shown in grey). They decide whether dendrite branches (DBL), soma (SL) and axo-synaptic branches (ASL) should die, change, or replicate. The remaining evolved computational function (WP) adjusts synaptic and dendritic weights and is used to decide the transfer of potential from a firing neuron (dashed line emanating from soma) to a neighbouring neuron

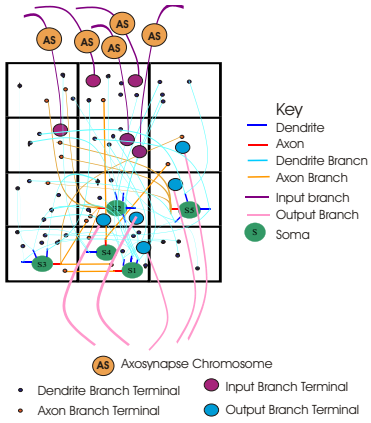

Figure 3: A schematic illustration of a 3×4 CGPCN grid. The grid contains five neurons, each neuron has a number of dendrites with dendrite branches, and an axon with axon branches. Inputs are applied at five random locations in the grid using input axo-synapse branches by running axo-synaptic CGP programs. Outputs are taken from five random locations through output dendrite branches. The figure shows the exact locations of neurons and branches as used in most of the experiments as an initial network. Each gird square represents one location, branches and soma are shown spaced for clarity. Each branch location is represented by where its terminal is located. Every location can have as many neurons and branches as the network produces, there is no imposed upper limit.

to other neurons through axo-synaptic branches. The same process is repeated in all neurons. A description of the seven chromosomes is given in the next section.

6.2 CGP Model of Neuron

In our model neural functionality is divided into three major categories: electrical processing, life cycle and weight processing. These categories are described in detail below.

Electrical Processing

The electrical processing part is responsible for signal processing inside neurons and communication between neurons. It consists of dendrite branch (D), soma (S), and axo-synaptic (AS) branch electrical chromosomes (as shown in figure 2).

D handles the interaction of dendrite branches belonging to a dendrite. It takes active dendrite branch potentials and soma potential as input and the updates their values. The *Statefactor* is decreased if the update in potential is large and vice versa. If a branch is active its life cycle program is run (DBL), otherwise it continues processing the other dendrites.

S, determines the final value of soma potential after receiving signals from all the dendrites. The processed potential of the soma is then compared with the threshold potential of the soma, and a decision is made whether to fire an action potential or not. If it fires, it is kept inactive (refractory period) for a few cycles by changing its *statefactor*, the soma life cycle chromosome (SL) is run, and the firing potential is sent to the other neurons by running the program encoded in axo-synapse electrical chromosome (AS). The threshold potential of the soma is adjusted to a new value (maximum) if the soma fires.

The potential from the soma is transferred to other neurons through axon branches. The AS program updates neighbouring dendrite branch potentials and the axo-synaptic potential. The *statefactor* of the axo-synaptic branch is also updated. If the axo-synaptic branch is active its life cycle program (ASL) is executed.

After this the **weight processing chromosome (WP)** is run which updates the *Weights* of branches in the same grid square. The processed axo-synaptic potential is assigned to the dendrite branch having the *largest* updated *Weight*.

Life Cycle of Neuron

This part is responsible for replication or death of neurons and neurite (dendrites and axon) branches and also the growth and migration of neurite branches. It consists of three life cycle chromosomes responsible for the neuron and neurites development.

The dendrite (DBL) and axo-synaptic (ASL) branch chromosomes update *Resistance* and *Health* of the branch. Change in *Resistance* of a neurite branch is used to decide whether it will grow, shrink, or stay at its current location. The updated value of neurite branch *Health* decides whether to produce offspring, to die, or remain as it was with an updated *Health* value. If the updated *Health* is above a certain threshold it is allowed to produce offspring and if below certain threshold, it is removed from the neurite. Producing

offspring results in a new branch at the same CGPCN grid point connected to the same neurite (axon or dendrite).

The soma life cycle (SL) chromosome produces updated values of *Health* and *Weight* of the soma as output. The updated value of the soma *Health* decides whether the soma should produce offspring, should die or continue as it is. If the updated *Health* is above certain threshold it is allowed to produce offspring and if below a certain threshold it is removed from the network along with its neurites. If it produces offspring, then a new neuron is introduced into the network with a random number of neurites at a different random location. This neuron is placed at a pseudo-random location.

7. EXPERIMENTAL SETUP

The experiment is organized such that an agent is provided with CGPCN as its computational network. It is allowed to play five games against a minimax based checker program (MCP) (in the non co-evolutionary case). The initial population is five agents each starting with a small randomly generated initial network and randomly generated genotypes. In each subsequent game of the five, the agent starts with a developed network from previous game. The genotype corresponding to the agent with the highest average fitness at the end of five games is selected as the parent for the new population. Four offspring formed by mutating the parent are created. Any learning behaviour that is acquired by an agent is obtained through the interaction and repeated running of program encoded by the seven chromosomes within the game scenario.

The MCP always plays the first move. The updated board is then applied to an agent's CGPCN. The potentials representing the state of the board are applied to CGPCN using the axo-synapse(AS) chromosome. The agent CGPCN is run which decide about its move. The game continues until it is stopped. It is stopped if either the CGPCN of an agent or its opponent dies (i.e. all its neurons or neurites dies), or if all its or opponent players are taken, or if the agent or its opponent can not move anymore, or if the allotted number of moves allowed for the game have been taken.

In the second set of experiments (co-evolution) both the agents in a game develop. As both the agents start with a random network, so both of them play random moves to start with, but the level of play improves as the game progresses, because their network develops during the course of the game. Each agent play five games while developing using a particular genotype. Each population consists of five genotypes and the agent on either side play against the best genotype of the opponent from the previous generation. The selection from generation to generation is based on how much an agent improves during the course of five games. We have devised the fitness function so that agents that play better on later games in the five game series receive a larger fitness score than the agents who play more poorly in later games. This was designed so that agents which learn during the series of five games are positively selected.

7.1 Inputs and outputs of the System

Input is in the form of board values, which is an array of 32 elements, with each representing a playable board square. Each of the 32 inputs represents one of the following five different values depending on what is on the square of the board (represented by I). Zero means empty square. $I = M = 2^{32} - 1$ means a king, (3/4)M means a piece, (1/2)M an opposing piece and (1/4)M an opposing king.

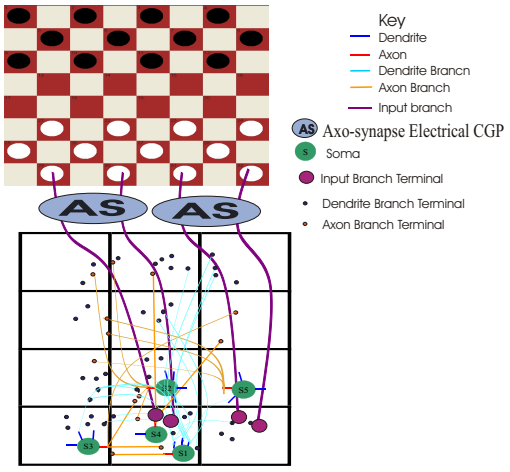

Figure 4: Interfacing CGPCN with Checker board. Four board positions are interfaced with the CGPCN such that board positions are applied in pair per square of CGPCN.

The board inputs are applied in pairs to all the sixteen locations in the 4x4 CGPCN grid (i.e. two input axo-synapse branches in every grid square, one axo-synapse branch for each playable position) as the number of playable board positions are 32 as shown in figure 4. Figure 4 shows how the CGPCN is interfaced with the game board, input axo-synapse branches are allocated for each playable board position. These inputs run programs encoded in the axo-synapse electrical chromosome to provide input into CGPCN (i.e. the axo-synapse CGP updates the potential of neighbouring dendrite branches).

Input potentials of the two board positions and the neighbouring dendrite branches are applied to the axo-synapse chromosome. This chromosome produces the updated values of the dendrite branches in that particular CGPCN grid square. In each CGPCN grid square there are two branches for two board positions. The axo-synapse chromosome is run for each square one by one, starting from square one and finishing at sixteenth.

Output is in two forms, one of the outputs is used to select the piece to move, and second is used to decide where that piece should move. Each piece on the board has an output dendrite branch in the CGPCN grid. All pieces are assigned a unique ID, representing the CGPCN grid square where its branch is located. So the twelve pieces of each player are located at the first twelve grid squares. The player can only see its pieces, while processing a move and vice versa. Also the location of output dendrite branch does not change when a piece is moved, the ID of the piece represent the branch location not the piece location. Each of these branches has a potential, which is updated during CGPCN processing. The values of potentials determine the possibility of a piece to move, the piece that has the highest potential will be the one that is moved, however if any pieces are in a position to jump then the piece with the highest potential of those will move. Note that if the piece is a king and can jump then, according to the rules of checkers, this takes priority. If two pieces are kings, and each could jump, the king with

the highest potential makes the jumping move. In addition, there are also five output dendrite branches distributed at random locations in the CGPCN grid. The average value of these branch potentials determine the direction of movement for the piece. Whenever a piece is removed its dendrite branch is removed from the CGPCN grid.

7.2 CGP Computational Network (CGPCN) Setup

The CGPCN is arranged in the following manner for this experiment. Each player CGPCN has neurons and branches located in a 4x4 grid. Initial number of neurons is 5. Maximum number of dendrites is 5. Maximum number of dendrite and axon branches is 200. Maximum branch *statefactor* is 7. Maximum soma *statefactor* is 3. Mutation rate is 5%. Maximum number of nodes per chromosome is 200. Maximum number of moves is 20 for each player.

7.3 Fitness Calculation

The fitness of each agent is calculated at the end of the game using the following equation:

$Fitness = A + 200(K_P - K_O) + 100(M_P - M_O) + N_M$,

Where K_P represents the number of kings, and M_P represents number of men (normal pieces) of the player. K_O and M_O represent the number of kings and men of the opposing player. N_M represents the total number of moves played. A is 1000 for a win, and zero for a draw. To avoid spending much computational time assessing the abilities of poor game playing agents we have chosen a maximum number of moves. If this number of moves is reached before either of the agents win the game, then A =0, and the number of pieces and type of pieces decide the fitness value of the agent.

8. RESULTS AND ANALYSIS

In two independent evolutionary runs we evolved agents against MCP (evolution) and co-evolved agents for one thousand (1000) generations. Then we took the best players from generations 50 to 1000 (in 50 generation intervals) from the co-evolutionary runs and let them play against the players evolved against the MCP at the same generation. In this way we could assess whether co-evolved players play better at the same generation than the agents that played only against the professional checker software (whose level of play does not change during the course of game). We evaluate their performance over the five game series by calculating their average fitness using the fitness function that was used in evolution. It is important to note that over the five game series *there is no evolution*. We just begin with a small random network and run the programs that were evolved at the generation in question over the sequence of five games.

In Figure 5 we have plotted the average fitness of both co-evolved and evolved player when playing each other in a five game series for different generations. The co-evolved player in almost every case beats the evolved player by a large margin. We also repeated these experiments under exactly the same conditions but where the players played a ten game sequence of games. In Figure 6 we have plotted the average fitness in the same way as before. Comparing the two figures, reveals that the players that were obtained through co-evolution perform even better than the five game players against the same players evolved against the MCP. This indicates that on average the players who play a ten

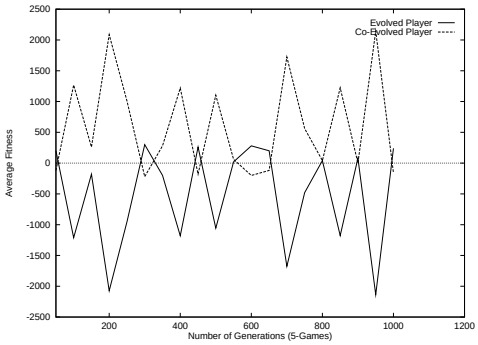

Figure 5: Average fitness of Co-evolved player against an evolved player for five games

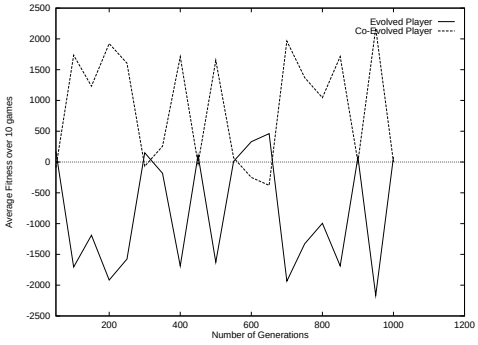

Figure 6: Average fitness of Co-evolved player against an evolved player for ten games

game series play checkers at a higher level than the players who play the five game series. It is important to note that the evolved programs for both cases are the same, the only difference is that in one case the programs play a series of ten games and the other they played only five games. This is strong evidence that the programs are actually learning how to play checkers better through experience alone.

To asess how large these margins of victory were, we plotted the cumulative fitness (where each plotted fitness is added to the previous) for both the agents playing a five game series and those playing a ten game series. This is shown in Figure 7 and we have plotted what the cumulative fitness would be if the co-evolved agents won every game against the evolved agents with one piece advantage, two pieces advantage (or one King), three pieces (a king and a piece), eight pieces (4 kings) or ten pieces (5 kings) advantage. From these graphs, it is evident that the co-evolved agent continues to perform better and wins every game by a margin greater than nine pieces on average. In fact, the ten game co-evolved players almost always beat the MCP evolved players by more than eight pieces (4 kings), whereas the five game players win by more than five pieces, but less than six. The figure also shows that the players with ten games experience are much superior to the same starting players but who have only five game experience.

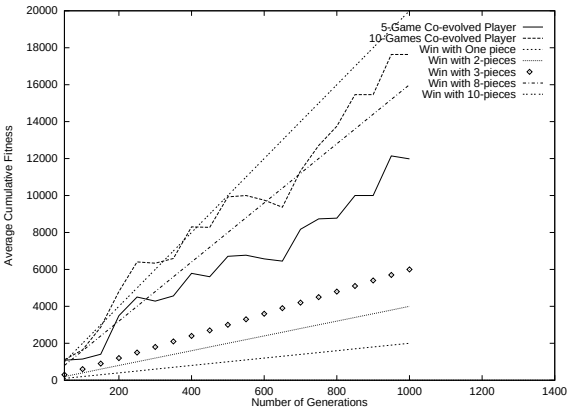

Figure 7: Average cumulative fitness of Co-evolved player against an evolved player for five and ten games

9. CONCLUSION

We have investigated the evolution and co-evolution of checkers playing agents that are controlled by developmental programs. The agents evolve intelligent behaviour much quicker through co-evolution rather than evolution against a minimax based program. We also have shown that the co-evolved agents improve with experience, and it appears that we have successfully evolved CGP programs that encode *an ability* to learn 'how to play' checkers. In future, we are planning to coevolve agents for longer, and allow more developmental experience through longer sequence of games, after evolution is finished.

10. REFERENCES

[1] A. Cangelosi, S. Nolfi, and D. Parisi. Cell division and migration in a 'genotype' for neural networks. *Network-Computation in Neural Systems*, 5:497–515, 1994.

[2] F. Dalaert and R. Beer. Towards an evolvable model of development for autonomous agent synthesis. In *Brooks, R. and Maes, P. eds. Proceedings of the Fourth Conference on Artificial Life*. MIT Press, 1994.

[3] R. Dawkins and J. R. Krebs. Arms races between and within species. In *Proceedings of the Royal Society of London Series B*, volume 205, page 489Ú511, 1979.

[4] D. Federici. Evolving developing spiking neural networks. In *Proceedings of CEC 2005 IEEE Congress on Evolutionary Computation*, pages 543–550, 2005.

[5] D. Fogel. *Blondie24: Playing at the Edge of AI*. Academic Press,London, UK, 2002.

[6] F. Gruau. Automatic definition of modular neural networks. *Adaptive Behaviour*, 3:151–183, 1994.

[7] W. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Artificial life 2*, pages 313–324, 1991.

[8] N. Jacobi. *Harnessing Morphogenesis, Cognitive Science Research Paper 423, COGS*. University of Sussex, 1995.

[9] G. Kendall and G. Whitwell. An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In *IEEE. CEC. 2001*, pages 995–1002, 2001.

[10] G. Khan, J. Miller, and D. Halliday. Coevolution of intelligent agents using cartesian genetic programming. In *Proc. GECCO*, pages 269 – 276, 2007.

[11] A. Lubberts and R. Miikkulainen. Co-evolving a go-playing neural network. *in Coevolution: Turning Adaptive Algorithms upon Themselves, Belew R. and Juille H (eds.)*, pages 14–19, 2001.

[12] J. Miller, D. Job, and V. Vassilev. Principles in the evolutionary design of digital circuits – part i. *Journal of Genetic Programming and Evolvable Machines*, 1(2):259–288, 2000.

[13] J. F. Miller and P. Thomson. Cartesian genetic programming. In *Proc. EuroGP*, volume 1802 of *LNCS*, pages 121–132, 2000.

[14] D. Moriarty and R. Miikulainen. Discovering complex othello strategies through evolutionary neural networks. *Connection Science*, 7(3-4):195–209, 1995.

[15] S. Nolfi and D. Floreano. Co-evolving predator and prey robots: Do 'arm races' arise in artificial evolution? *Artificial Life*, 4:311–335, 1998.

[16] S. Nolfi, O. Miglino, and D. Parisi. Phenotypic plasticity in evolving neural networks. in gaussier, d.p, and nicoud, j.d., eds. In *Proceedings of the International Conference from perception to action*. IEEE Press, 1994.

[17] J. Paredis. Coevolutionary constraint satisfaction. In *Proceedings of the third international conference on parallel problem solving from nature, Springer- Verlag*, volume 866, pages 46–55, 1994.

[18] J. Paredis. Coevolutionary computation. *Artificial Life*, 2(4):355–375, 1995.

[19] J. Pollack, A. Blair, and M. Land. Coevolution of a backgammon player. In *In: Langton, C. (ed),Proceedings artificial life 5*. MIT Press.

[20] D. Roggen, D. Federici, and D. Floreano. Evolutionary morphogenesis for multi-cellular systems. *Journal of Genetic Programming and Evolvable Machines*, 8:61–96, 2007.

[21] C. D. Rosin. *Coevolutionary search among adversaries*. Ph.D. thesis, University of California, San Diego., 1997.

[22] A. Rust, R. Adams, and B. H. Evolutionary neural topiary: Growing and sculpting artificial neurons to order. In *Proc. of the 7th Int. Conf. on the Simulation and synthesis of Living Systems (ALife VII)*, pages 146–150. MIT Press, 2000.

[23] A. G. Rust, R. Adams, S. George, and H. Bolouri. Activity-based pruning in developmental artificial neural networks. In *Proc. of the European Conf. on Artificial Life (ECAL'97)*, pages 224–233. MIT Press, 1997.

[24] J. Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer, Berlin, 1996.

[25] G. Shepherd. *The synaptic organization of the brain*. Oxford Press, 1990.

[26] A. Van Ooyen and J. Pelt. Activity-dependent outgrowth of neurons and overshoot phenomena in developing neural networks. *Journal of Theoretical Biology*, 167:27–43, 1994.