Developing Neural Structure of Two Agents that Play Checkers Using Cartesian Genetic Programming

Gul Muhammad Khan
Electronics Department
University of York
York, YO10 5DD,UK
gk502@ohm.york.ac.uk

Julian F. Miller
Electronics Department
University of York
York, YO10 5DD,UK
jfm7@ohm.york.ac.uk

David M. Halliday
Electronics Department
University of York
York, YO10 5DD,UK
dh20@ohm.york.ac.uk

ABSTRACT

A developmental model of neural network is presented and evaluated in the game of Checkers. The network is developed using cartesian genetic programs (CGP) as genotypes. Two agents are provided with this network and allowed to co-evolve untill they start playing better. The network that occurs by running theses genetic programs has a highly dynamic morphology in which neurons grow, and die, and neurite branches together with synaptic connections form and change in response to situations encountered on the checkers board. The method has no board evaluation function, no explicit learning rules and no human expertise at playing checkers is used. The results show that, after a number of generations, by playing each other the agents begin to play much better and can easily beat agents that occur in earlier generations. Such learning abilities are encoded at a *genetic* level rather than at the phenotype level of neural connections.

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

Ever since the beginnings of the field of Artificial Intelligence research, building computer programs that play games has been considered a worthwhile objective. Indeed, as early as 1950, Claude Shannon published an influential paper on computer chess [15]. Shannon developed the idea of using a game tree of a certain depth and advocated using a *board evaluation function*. This function allocates a numerical score according to how good a board position is for a player. The method of minimax was developed by a number of people and has a complicated genesis [3], although it is often attributed to von Neumann [11]. Arthur Samuel presented a seminal paper in 1959 on computer checkers [12] in which he refined, during play, a board evaluation that was a weighted sum of various factors that are important in determining the favourability of a board position for a player. Samuel used the board evaluation function to determine a numerical value for a board position and then applied minimax to evaluate a game tree of a particular depth. His method was 'self-learning', since after two computer players have played a game, the loser is replaced with a deterministic variant of the winner by altering the weights on the features that were used, or in some cases replacing features that had very low weight with other features. Nowadays, the game of checkers is considered solved [14]. The current world champion at checkers is a computer program called Chinook[13]. This program is mostly based on a linear handcrafted evaluation function that considers several features of the game board including: 1) piece count, 2) kings count, 3) trapped kings, 4) turn, 5) runaway checkers and other minor factors. In addition the program has: 1) access to a library of opening moves from games played by grand masters, 2) the complete endgame database for all boards with eight or fewer pieces. The program is completely based on human knowledge and no machine learning methods are used in its development.

Chellapilla and Fogel produced a checker playing program called Anaconda [4]. They used co-evolution of artificial neural networks (ANNs) and were able to evolve an ANN that could play at master level. The role of the ANN was to provide a board evaluation function in a minimax procedure.

Although the history of research in computers playing games is full of outstanding and highly effective methods none of them bear much resemblance to the methods that human beings appear to use to play games well. Firstly, human beings do not employ minimax and secondly, they do not use a numerical board evaluation function. Typically they consider relatively few potential board positions and evaluate the favourability of these boards in a highly intuitive and heuristic manner. They usually learn during a game, indeed this is how, generally humans learn to be good at any game. So the question arises: How is this possible? Can a method be devised in which a computer plays a game *without* having a board evaluation function? Obvi-

GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.
ACM 978-1-60558-131-6/08/07.

ously it needs a method that decides a move, but this does not have to mean that such a method has to use a function that compares the favourability of whole collection of possible board positions each time a move should be made. This is the approach we have taken. In our work we are interested in how an *ability to learn* can arise and be encoded in a genotype that *when executed* gives rise to a neural network that can play a game well. The genotype we evolve is a set of computational functions that represent various aspects of biological neurons [6]. Each agent (player) has a genotype that grows a computational neural structure and through co-evolution, the developed structure allows 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. Instead, all aspects of neural functions are obtained *ab initio* through evolution of the genotype.

Section 2 gives an overview of Cartesian Genetic Programming, section 3 provide an overview of neural development, section 4 describes the structure and operation of our computational network, section 5 describes our results from the co-evolution of two agents playing each other and section 6 provides some concluding remarks.

2. CARTESIAN GENETIC PROGRAMMING (CGP)

Cartesian Genetic Programming, which developed from the work of Miller and Thomson [10, 9], represents programs by directed acyclic graphs. CGP use a rectangular grid of computational nodes, but the number of rows can be one (as used in this paper). The genotype is a fixed length list of integers, which encode the function of nodes and the connections of the directed graph. The nodes can take their inputs from either the output of a previous node or from a program input (terminal). The number of inputs that a node has is dictated by the number of inputs that are required by the function it represents. The phenotype is obtained by following the connected nodes from the program outputs to the inputs. In this process, some node outputs may not be used so that their genes have no influence on the final decoded program. Such non-coding genes have no effect on genotype fitness.

3. NEURAL DEVELOPMENT

Multicellular biological systems are built through a developmental process from relatively simple gene structures. The same technique could be used in computational development to produce complex systems from simple systems that are capable of learning and adapting. Phenotypes are developed through the interaction of genes at different hierarchical levels, this provides the capability of self-organization, which, for example, can be seen in an ant colony [5], [2].

It is well known that evolutionary algorithms can get stuck at local optima, and measures used to counter this, such as increasing mutation rates, are only useful at early stages of evolution when solutions are not very fit. However, small mutational changes of developmental systems can sometimes completely change the kind of phenotype developed. This can alleviate the problems of being trapped.

Most ANN models ignore the fact that neurons are part of a phenotype which is derived from the genotype through a process called development [7] (page 339-40). The information in the genotype specifies the rules for developing the nervous system, based on environmental interaction during the developmental phase.

4. THE CGP COMPUTATIONAL NETWORK (CGPCN)

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

The CGPCN network has two main aspects:

- Neurons with dendrites, dendrite branches, and an axon with axon branches.
- A genotype of seven chromosomes representing the genetic code of each neuron.

The first aspect defines the neural components and their properties, and the second is concerned with the internal behaviour of the neurons in the network. Each genotype consists of seven chromosomes, each represented as a digital circuit using CGP. These chromosomes represent the functionality of different parts of the neuron. During evolution the second aspect(genotype) is evolved so that the phenotype that results has the best functionality, whereas the first aspect (the neural components and their properties) only change during the life time of the network, i.e while it is performing the learning task.

The CGPCN is organized in such a way that neurons are placed randomly in a two dimensional grid (the CGPCN grid) so that they are only aware of their spatial neighbours (as shown in figure 1). The initial number of neurons is specified by the user. Initially, each neuron is allocated a random number of dendrites, and 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. Branches may grow or shrink and thus move from one CGPCN grid point to another, They can produce new branches, and can disappear. Neurons may produce new daughter neurons, or may die. Axon branches transfer information only to dendrite branches in their proximity.

Information processing in the network starts by selecting the list of *active* neurons in the network and processing them in a random sequence. This sequence is called a *cycle*. The processing of neural components is carried out in time-slices so as to emulate parallel processing. Each neuron takes the signal from its dendrites by running the dendrite electrical processing chromosomal program. The signals from dendrites are averaged and applied together with the existing soma potential to the soma program. This is run to get the final value of soma potential, which is used to decide whether it will fire or not. If it fires the action potential signal is transferred to other neurons through axosynaptic branches. The same process is repeated in all neurons. The functionality of each neuron is determined by the seven CGP chromosomes (described below).

In addition we have included a number of specific parameters for different components. These are the *statefactor*, *health*, *weight* and *resistance*. These are described below, see also figure 3. When a neural component's *statefactor* is zero it is considered to be active and the corresponding genetic program is run. The value of the *statefactor* is affected by CGP programs. After each neural network cycle

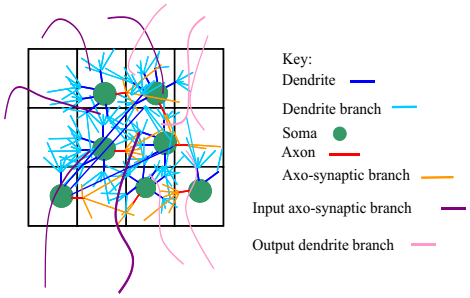

Figure 1: A schematic illustration of a 3×4 CGPCN grid. The grid contains seven neurons, each neuron has a number of dendrites with dendrite branches, and an axon with axon branches. Inputs are applied in the grid using virtual axons. Outputs are taken through virtual dendrite branches. Note that the system does not distinguish relative locations *within* each grid point, the fine detail is included for clarity of illustration only.

the potential of the soma and the branches are reduced by certain factor(2%) and the *statefactor* is decremented. The reduction in potential occurs to emulate the natural decay of action potential voltages in real neurons, and the reduction in *statefactor* is there to make inactive branches and neurons move towards activity. After a user defined number of cycles(5 in this case) of the network, the *health* and *weight* of neurons and branches are also reduced by certain factor(2%). When the health of a neuron or branch falls below a certain threshold(10%), it dies and is removed from the network. The *health* parameter was included so that there would be tendency for neurons and branches to fade away and evolution is responsible for producing programs that maintain stability against this background decay process. The *resistance* is a parameter that is used to control growth and/or shrinkage of dendrites and axons.

4.1 CGP Model of Neuron

Each chromosome is a digital circuit that operates on 32-bit binary numbers. Individual CGP nodes consist of one of four 2 to 1 multiplexer operations [8], that are obtained by successive inversion of one or both data inputs. The seven chromosomes that define neural functionality are divided into three categories; Electrical Processing (3), Life Cycle Processing (3) and Weight Processing (1).

4.1.1 Electrical Processing

The electrical processing chromosomes are responsible for signal processing inside each neuron and for communication between neurons. They represent respectively, dendrites, soma and axons (as shown in Figure 2).

Electrical Processing in Dendrite This handles the interaction between potentials of different dendrite branches belonging to the same dendrite. Figure 2 shows the inputs and outputs. The input consists of potentials of all the *active* branches connected to the dendrite and the soma potential. The CGP program produces the new values of the dendrite branch potentials as output. The potential of each branch is processed by adding weighted values of *resistance*, *health*, and *weight* of the branch. The *statefactor* of each branch is adjusted based on the updated value of branch potential.

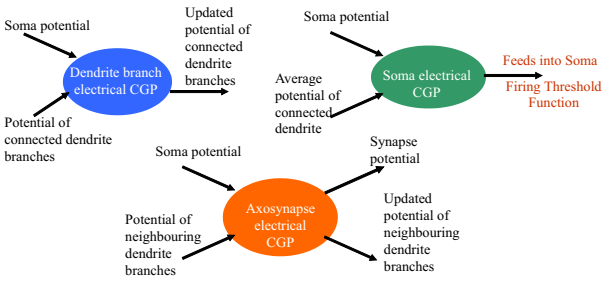

Figure 2: Schematic diagram of the three electrical processing CGP programs showing inputs and outputs: Dendrite branch, Soma and Axo-synapse.

Electrical Processing in Soma This is responsible for determining the value of soma potential after receiving signals from all the dendrites. A two stage averaging process is used to arrive at the scalar potential value used as input to the soma CGP program:

1. The potential in each dendrite is calculated as the average of the potential from all daughter branches of that dendrite.

2. The potential input to the soma is calculated as the average potential of all dendrites in the neuron.

The input potential along with the existing soma potential are applied as the two scalar inputs to the chromosome as shown in Figure 2. The chromosome produces an updated value of the soma potential. This is further processed using a weighted sum that incorporates the *health* and *weight* of the soma.

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 the soma fires it is kept inactive (refractory period) for a number cycles by changing its *statefactor*, then the soma life cycle chromosome is run, and the output potential is signalled to other neurons by running the axosynapse electrical processing chromosome. The threshold potential of the soma is adjusted to a new value if the soma fires.

Electrical Processing in Axo-Synaptic Branch Figure 2 shows the inputs and outputs to the chromosome responsible for the electrical processing in each axosynaptic branch.

This chromosome produces the updated values of dendrite branch potentials and the axo-synaptic potential as output (see figure 2). The axo-synaptic potential is further processed as a weighted sum incorporating the *health*, *weight* and *resistance* of the axo-synaptic branch. Then the axo-synaptic branch weight processing program is run (see figure 4) and the processed axo-synaptic potential is assigned to the dendrite branch having the highest updated *weight*, replacing its previously updated potential. This emulates how biological neurons communicate electrically. After this, the *statefactor* of the axosynaptic branch is also updated. If the axo-synaptic branch is active its life cycle program is executed.

4.1.2 Life Cycle of Neuron

This section summarises the life cycle activities in the network. The life cycle is responsible for increases or de-

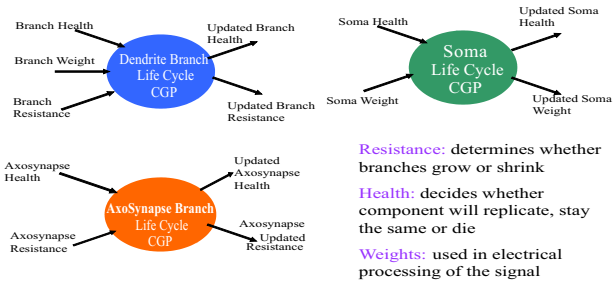

Figure 3: Life cycle of neuron, showing CGP programs for life cycles in Dendrite branch, Soma, and Axosynapse branch with their corresponding inputs and outputs.

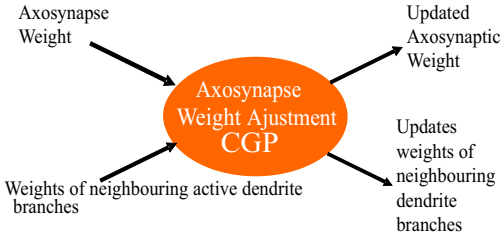

Figure 4: Schematic drawing of weight processing CGP program for axo-synaptic branches, showing inputs and outputs.

creases in the numbers of neurons, dendrite branches and axon branches, and for the growth and migration of dendrite and axon branches. It consists of three chromosomes that provide the life cycle programs for the dendrite branch, the soma, and the axo-synapse branch.

Life Cycle of Dendrite Branch This chromosome controls the life cycle of dendrite branches. Figure 3 shows the inputs and outputs to the chromosome. Variation in *resistance* of a dendrite branch is used to decide whether it will grow, shrink, or stay at its current location.

The updated value of dendrite branch *health* is used to decide whether to a branch produces offspring, dies, or remains unchanged (but with an updated *health* value). An offspring is a new branch at the same CGPCN grid point connected to the same dendrite.

Life Cycle of Soma Figure 3 shows the inputs and outputs of the soma life cycle chromosome. This chromosome evaluates the life cycle of the neuron. This chromosome produces updated values of *health* and *weight* of the soma as output. The updated value of *health* decides whether the soma should produce offspring, should die or continue without change.

Axo-Synaptic Branch Life Cycle Figure 3 shows the inputs and outputs of the axosynaptic branch life cycle chromosome. It takes as input the *health* and *resistance* of the axon branch, and generates updated values as output.

The updated values of *resistance* are used to decide whether the axon branch should grow, shrink, or stay at its current location. The *health* of the axon branch decides whether the branch will die, produce offspring, or merely continue with an updated value of *health*.

4.1.3 *Weight Processing*

Weight processing is responsible for updating the *weights* of axo-synaptic and dendrite branches. It consists of the axo-synaptic branch *weight* processing chromosome. The *weight* of axon and dendrite branches affect their capability to transfer information efficiently. The *weights* are responsible for modulating the signal. They affect almost all the neural processes, either by virtue of being an input to a chromosomal program, or as a factor in post processing of signals.

Figure 4 shows the inputs and the outputs to the axo-synaptic weight processing chromosome. The CGP program encoded in this chromosome takes as input the *weight* values of the axo-synapse and the *weight* values of dendrite branches from the same CGPCN grid square, and produces updated *weight* values as output.

4.2 Inputs and Outputs

This section describes how inputs are applied to the overall network and outputs are obtained from the network. Inputs are applied to the CGPCN through "virtual" axon branches by using the axo-synaptic electrical processing chromosome. These virtual branches are distributed in the network in a similar way to the axon branches of neurons as shown in figure 1. They take input from the environment and transfer it through virtual axo-synapses without processing it. When inputs are applied to the system, the program encoded in the axo-synaptic electrical processing chromosome is executed, and the resulting signal is transferred to neighbouring active dendrite branches.

Similarly, the signals from the system are read out through virtual dendrite branches. These virtual dendrite branches are distributed across the network as shown in Figure 1. These branches are updated by the axo-synaptic electrical processing chromosome in the same way as other dendrite branches. The output from this is taken without further processing after every five cycles.

4.3 Fitness Calculation and Evolutionary Strategy

The fitness of the agents is accumulated at the end of every game using the following equation:

$$Fitness = A + 200N_K + 100N_M - 200N_{OK} - 100N_{OM} + N_{MOV}$$

Where N_K represents the number of kings, and N_M represents number of men of the current player. N_{OK} and N_{OM} represent the number of kings and men of the opposing player. N_{MOV} 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.

The evolutionary strategy [1] utilized is of the form $1 + \lambda$, with λ set to 4 [16], i.e. one parent with 4 offspring (population size 5). The parent, or elite, is preserved unaltered, whilst the offspring are generated by mutation of the parent. The best chromosome is always promoted to the next generation, however, if two or more chromosomes achieve the highest fitness then the newest (genetically) is always chosen [8].

5. THE GAME: CO-EVOLUTION OF TWO AGENTS PLAYING CHECKERS

Each agent is provided with a CGPCN 'brain', and plays checkers against the other. Each agent's population consists of five genotypes. During every generation the genotype of an agent is chosen from the best performing of five genotypes. Each of the five first agent population members are tested against the best performing second agent genotype from the previous generation. Similarly each of the five second agent population members are tested against the best performing first agent genotype from the previous generation. The initial random network is the same for both the first and second agent. It is the genotypes which grow a mature network from the initial randomly generated network. The best first and second agent genotypes are selected as the parents for the new populations and are promoted to the next generation unaltered along with four offspring (mutational variants of the best performing agents). The performance of the agent is calculated only at the end of every game. The CGPCN structure for the two agents develop a lot during the game, while, of course, the genotype remains the same, and is only changed from generation to generation. The initial CGPCN setup remains the same during the course of evolution. So all the agents start with the same initial random CGPCN structure. Thus any learning behaviour that exists in an agent is obtained through the interaction and repeated running of the seven chromosomes in the game scenario.

The initial CGPCN structure starts with five neurons with a random number of dendrites, one axon and a random number of dendrite and axon branches. These neurons are located in a grid. The neurons are placed in close proximity to each other so that they can more easily communicate with each other.

When the experiment starts, the agent playing black takes input from the board. This input is applied to its CGPCN through virtual axosynapses. The CGPCN network is then run for five cycles. During this process it updates the potentials of the virtual dendrite branches acting as the output of the network. These updated potentials are averaged, and used to decide the direction of movement for the corresponding piece. Each piece is allocated a virtual dendrite branch in the CGPCN (see later). The potentials of these branches are updated during CGPCN process. The updated values of these potentials are used to decide which piece to move, unless there is a jump, which takes priority. For more than one jump, the piece with highest potential makes the jump. The same process is repeated for the opponent and the process is repeated and continues until the game stops.

The game 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.

5.1 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 15. 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.

5.2 Inputs and outputs of the System

The input is applied to both CGPCNs using virtual axosynapse branches. 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. A zero value indicates an empty square. A maximum value of $2^{32} - 1$ represents a king of the player making the move. Three quarters of the maximum value represents an ordinary piece of the player making the move. Half the maximum value represents an opposing player's piece. Quarter the maximum value represents an opponent's king. The board inputs are applied in pairs to all the sixteen locations in the 4x4 CGPCN grid (i.e. two virtual axo-synapse branches in every grid square).

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 a virtual dendrite branch in the CGPCN. All pieces are assigned a unique ID, representing the CGPCN grid square where its branch is located. 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. Once again if two pieces are kings and each could jump the the king with the highest potential makes the jumping move. In addition, there are also five virtual 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.

5.3 Results and Analysis

We believe that checkers represents a difficult problem. The two agents each start with a few neurons with a random number of dendrites and branches, and with random connections. Evolution must first find a series of programs that build a computational network that is capable of solving the task while maintaining a stable network (i.e. not losing all the neurons or branches). Secondly, it must find a way of processing the environmental signals and differentiating among them. Thirdly, it must understand the spatial layout of the board (positions of its players). Fourth it must develop a memory or knowledge about the meaning of the signals from the board, and fifth it should develop a memory of all it previous moves and whether they were beneficial or deleterious. Finally it should understand the benefits of making a king or jumping over. Over the generations the agents learn from each other about favourable moves, this learning is transferred through the genes from generation to generation. Thus one would expect well evolved agent to play much better than earlier ones.

To test whether more evolved agents were indeed playing the game better, we did a number of experiments. We tested a well evolved agents against less evolved agents. We found

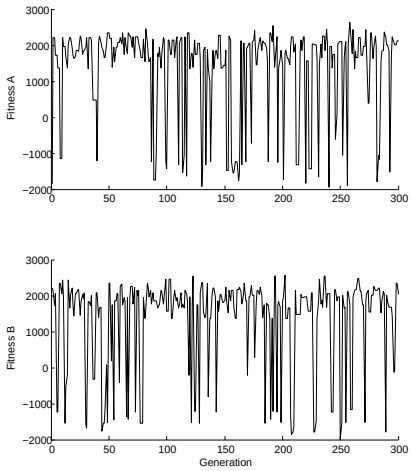

Figure 5: Variations in fitness of two agents over one evolutionary run.

that the well evolved agent always beat the less evolved one, in some of the cases it ends up in a draw, but in those cases the well evolved agent ends up with more kings and pieces than the less evolved agent. Table 1 shows the difference in fitness of the agents at different generations and their performance when playing each other.

Figure 5 shows the variation in the fitness of the two agents over a particular evolutionary run, the graph show a lot of variation from generation to generation. This is typical of co-evolution. In every generation agents try to beat the best opponent from the last generation, if it wins its fitness is increased, at the expense of the opponent. This causes the frequent fluctuations.

6. CONCLUSION

We have described a neuron-inspired developmental approach to construct a new kind of computational neural architectures. These control the actions of agents playing checkers. We found that the neural structures controlling the agents grow and change in response to their behaviour, interactions with each other and the environment. The evolved programs built neural structures from an initial small random structure. The structures develop during a single game, and allow them to learn and exhibit intelligent behaviour. We used a technique called Cartesian Genetic Programming to encode and evolve seven computational functions inspired by the biological neuron. In future work, we plan to evaluate this approach in richer and more complex environments. The eventual aim is to see if it is possible to evolve a general capability for learning.

White	Black	White Fitness	Black Fitness
1350	5	2148(8 men,1 king)	-852
5	1350	-551	2049(2 kings,4 men)
1750	150	680(2 kings,1 man)	80(1 king)
150	1750	-234	1766(2 kings,1 man)
50	500	281(1 kings,1 man)	481(2 king,1 man)
500	50	2147(3 kings,5 men)	-653
10	100	-221	1779(2 kings,3 men)
350	800	-259(3 men)	341 (6 men)
800	200	1441(2 kings,9 men)	-1359(1 men)
150	800	-459(2 men)	541(1 kings,4 men)

Table 1: The number of generations used to evolve agents and their fitnesses and performance when they played each other

7. REFERENCES

[1] T. Back, F. Hoffmeister, and H. Schwefel. A survey of evolution strategies. In *Proceedings of the 4th International Conference on Genetic Algorithms,Vol. 1802, Morgan Kaufmann*, pages 2–9, 1991.

[2] P. Bentley. *Digital Biology*. Simon and Schuster, 2002.

[3] R. W. Dimand and M. A. Dimand. *A History of Game Theory: From the Beginnings to 1945*, volume 1. Routledge, 1996.

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

[5] J. Holland. *Emergence: from chaos to order*. Oxford University Press, 1998.

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

[7] S. Kumar and J. Bentley. *On Growth, Form and Computers*. Academic Press, 2003.

[8] 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.

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

[10] J. F. Miller, P. Thomson, and T. C. Fogarty. Designing electronic circuits using evolutionary algorithms. arithmetic circuits: a case study. pages 105–131, 1997.

[11] J. v. Neumann. Zur theorie der gesellschaftsspiele. *Math. Annalen*, 100:295–320, 1928.

[12] A. Samuel. Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.*, 3(3):210–219, 1959.

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

[14] J. Schaeffer and J. v. d. Herik. *Chips Challenging Champions*. Elsevier, Amsterdam, 2002.

[15] C. Shannon. Programming a computer for playing chess. *Phil. Mag.*, 41:256–275, 1950.

[16] T. Yu and J. Miller. Neutrality and the evolvability of boolean function landscape. In *Proc. EuroGP*, pages 204–217. Springer-Verlag, 2001.