Coevolution of neuro-developmental programs that play checkers

Gul Muhammad Khan, Julian F.Miller, and David M.Halliday

Electronics Department, University of York, York, YO10 5DD,UK
{gk502,jfm,dh20}@york.ac.uk
http://www.york.ac.uk

Abstract. This paper presents a method for co-evolving neuro-inspired developmental programs for playing checkers. Each player's program is represented by seven chromosomes encoding digital circuits, using a form of genetic programming, called Cartesian Genetic Programming (CGP). The neural network that occurs by running the 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 results show that, after a number of generations, by playing each other the agents play much better than those from earlier generations. Such learning abilities are encoded at a *genetic* level rather than at the phenotype level of neural connections.

Key words: Cartesian Genetic Programming, Adaptive computing, Co-evolution of hybrid systems, Artificial Neural Networks

1 Introduction

Although the history of research in computers playing games is full of highly effective methods (e.g. minimax, board evaluation function), it is arguable that human beings use such methods. 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? 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. 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.

The computational network possessed by each agent is based on a compartmentalised model of neural functions inspired by neuroscience[2]. In the model we have idealised seven neural functions which we have encoded as chromosomes (as explained later). These represent various aspects of the neuron: soma, dendrites and axon branches, and synaptic connections. The collection of chromosomes (forming the genotype) is encoded and evolved using a well known GP technique, Cartesian Genetic Programming (CGP) [5, 4]. The neurons are placed in a two dimensional grid. Neurite branches are allowed to grow and shrink, and communicate with each other via synapses. Dendrites [6], synaptic dynamics [1] and synaptic communication have been included to enhance the capabilities of the computational network. The network we describe has the potential virtue that it is autonomous, in the sense that when the compartmentalised chromosomal programs are run, a network of neurons, neurites and synapses grows in response to its own internal dynamics and the agent's environmental experiences. Section 2 gives an overview of Cartesian Genetic Programming. Section 3 describes the structure and operation of our computational network. Section 4 describes our results and analysis and section 5 provides some concluding remarks.

2 Cartesian Genetic Programming (CGP)

CGP represents programs by directed acyclic graphs [5, 4]. The number of rows is chosen to be one, so that an arbitrary directed graph may be evolved. 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 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. The CGP function nodes we have used are variants of 2 to 1 multiplexers [3] in which data inputs are either inverted or not. The multiplexers require four genes each to describe which type of multiplexer and its connections. Currently in our model all multiplexers operate in a bitwise fashion on 32-bit data.

The data operations on genotypes are of two types: scalar processing or vector processing. In the former, the inputs and outputs are 32-bit integers while in the latter inputs required by the chromosome are arranged in the form of an array, which is then divided into 10 CGP input vectors. If the total number of inputs can't be divided into ten equal parts, then they are padded with zeros. This allows us to process an arbitrary number of inputs by the CGP circuit chromosome simply by clocking through the elements of the vectors.

The evolutionary strategy utilized is of the form $1 + \lambda$, with λ set to 4, 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, it is important to note that if two or more chromosomes achieve the highest fitness then the *newest* (genetically) is always chosen [3].

3 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 (each represented as a digital circuit). It is the genotypes which grow a mature network from the initial randomly generated network.

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. Initially all neuro-components are assigned random values of potentials, resistances, statefactors and weights.

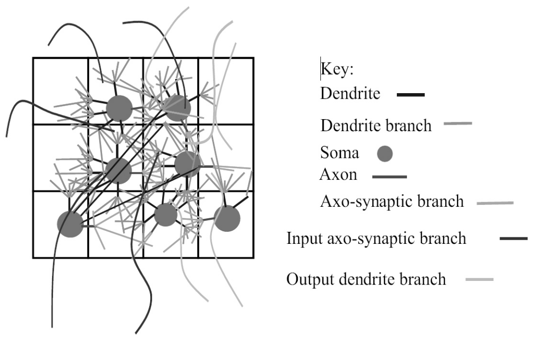


Fig. 1. A schematic illustration of a 3×4 CGPCN grid. The grid contains seven neurons, each neuron has a number of dendrites with dendrite branches, and an axon with axon branches. Inputs are applied in the grid using input axons. Outputs are taken through output dendrite branches. Note that the system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

Health, Resistance, Weight and Statefactor

Four 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 the neurons and branches are 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.

3.1 Inputs, Outputs and Information Processing in the Network

The external inputs (encoding a simulated potential) are applied to the CGPCN using input axo-synapse branches. These are distributed in the network in a similar way to the axon branches of neurons. They use axo-synapse electrical chromosome (explained later) to transfer signal to the neubouring dendrite branches. Similarly we have output dendrite branches. These branches are updated by the axo-synaptic chromosomes of neurons in the same way as other dendrite branches and, after every 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 using the circuit shown in figure 2. The processing of neural components is carried out in time-slices so as to emulate parallel processing. 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 (see Fig 2). 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 soma fires, an action potential is transferred to other neurons through axosynaptic branches. The same process is repeated in all neurons. After each cycle of neural network the potential of the soma and the branches are reduced by 10%. The state factor of soma and branches is also reduced by one unit after every cycle. This makes inactive branches and neurons move gradually towards activity. After a few cycles of network (typically 5), the *Health* and *Weights* of neurons and branches are also reduced by 10%, to facilitate the removal of unimportant neurons and branches. Description of the seven chromosomes is given in the next subsection.

3.2 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.

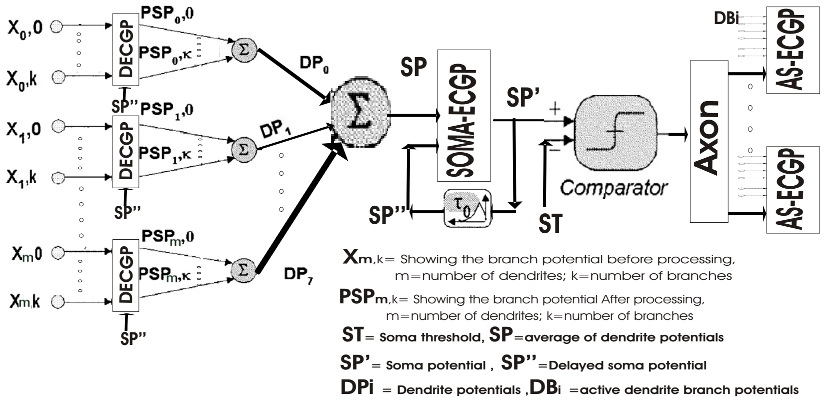

$X_0,0$
X_0,k
$X_1,0$
X_1,k
X_m0
X_mk
DECGP
DECGP
DECGP
$PSP_0,0$
PSP_0,K
$PSP_1,0$
PSP_1,K
$PSP_m,0$
PSP_m,K
SP"
SP"
SP"
DP_0
DP_1
DP_7
SP
SOMA-ECGP
SP'
SP"
τ_0
ST
Comparator
Axon
DBi
AS-ECGP
AS-ECGP
$X_{m,k}$= Showing the branch potential before processing, m=number of dendrites; k=number of branches
$PSP_{m,k}$= Showing the branch potential After processing, m=number of dendrites; k=number of branches
ST= Soma threshold, SP=average of dendrite potentials
SP'= Soma potential , SP"=Delayed soma potential
DPi = Dendrite potentials ,DBi =active dendrite branch potentials

Fig. 2. Electrical processing in neuron at different stages, from left to right branch potentials are processed by DECGP, then averaged at each dendrite, and soma, which processes it further using the Soma-ECGP giving a final soma potential. This is fed in to a comparator which decides whether to fire an action potential. This is transferred using the AS-ECGP

Electrical Processing

This part is responsible for signal processing inside neurons and communication between neurons. It consists of dendrite branch, soma, and axo-synaptic branch electrical chromosomes. The way they process electrical signal and transfer to other neurons is shown in Figure 2.

Dendrite Electrical CGP (DECGP) The DECGP vector chromosome handles the interaction between potentials of dendrite branches of a dendrite. The input consists of potentials of all the active branches connected to the dendrite and the soma potential. Since in practice there are many dendrite branch potentials and one soma potential, we increase the importance of the soma potential by creating multiple entries (equal to number of active dendrite branches) of it (in the input vector) before applying it to the DECGP, which 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* is decreased if the update in potential is large and vice versa. If any of the branches are active (has its statefactor equal to zero), its life cycle program is run, otherwise it continues processing the other dendrites.

Soma Electrical CGP (Soma-ECGP) The Soma-ECGP scalar chromosome determines the final value of soma potential after receiving signals from all the dendrites. The potentials of all dendrites are averaged, which in turn are the average of potentials of active branches attached to them (figure 2). This average potential and the soma potential is applied to the Soma-ECGP. It updates the value of the soma potential, which is further processed with a weighted summation of *Health* and *Weight* of the soma. The processed potential of the soma is then compared with the threshold potential of the soma (see Figure 2), 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 is run, and the firing potential is sent to the other neurons by running the Axo-Synapse Electrical CGP. The threshold potential of the soma is adjusted to a new value (maximum) if the soma fires.

Axo-Synaptic Electrical CGP (AS-ECGP) The potential from the soma is transferred to other neurons through axon branches. Both the axon and the synapse are considered as a single entity with combined properties. Figure 2 shows the inputs and outputs to the AS-ECGP vector chromosome. As mentioned before, the soma potential is biased (by introducing multiple entries of soma potential to increase its impact). AS-ECGP updates neighbouring dendrite branch potentials and the axo-synaptic potential. The axo-synaptic potential is then processed as a weighted summation of *Health*, *Weight* and *Resistance* of the axon branch. The *statefactor* of the axosynaptic branch is also updated. If the axo-synaptic branch is active its life cycle program is executed.

Weight Processing

The Weight processing CGP vector chromosome is responsible for updating the *Weights* of branches. The *Weights* of axon and dendrite branches affects their capability to modulate and transfer the information (signal) efficiently. 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. The CGP program encoded in this chromosome takes as input the *Weights* of the axo-synapse and the *neighbouring* (Same CGPCN grid square) dendrite branches and produces their updated values as output. The processed axo-synaptic potential(explained above) is assigned to the dendrite branch having the *largest* updated *Weight*.

Life Cycle of Neuron

This part is responsible for development of CGPCN. It consists of three scalar life cycle chromosomes.

Life Cycle of Dendrite and Axo-Synaptic Branches These two chromosome 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. If the change in *Resistance* during the process is above certain threshold the branch is allowed to migrate to a different neighbouring location at random. 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).

Life Cycle of Soma This scalar 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.

4 The Game: Co-evolution of two agents playing Checkers

Each agent is provided with a CGPCN, and plays checkers against the other. Each of the five first agent population members are tested against the best performing second agent genotype from the previous generation [7] and vice versa. The initial random network is the same for both the first and second agent. 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.

When the experiment starts, the agent playing black takes input from the board. This input is applied to its CGPCN through input axosynapses. The CGPCN network is then run for five cycles. During this process it updates the potentials of the output 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 output dendrite branch in the CGPCN (see later).

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.

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 10. 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.

Fitness Calculation

Both the agents are allowed to play a limited number of moves and the fitness of the agents is accumulated at the end this period 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. If the maximum 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.

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). The values taken by I are as follows: if empty I=0, if king I=Maximum value(M) $2^{32}-1$, if piece I=(3/4)M, if opposing piece I=(1/2)M, and finally if opposing king, I=(1/4)M. 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).

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 output 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 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.

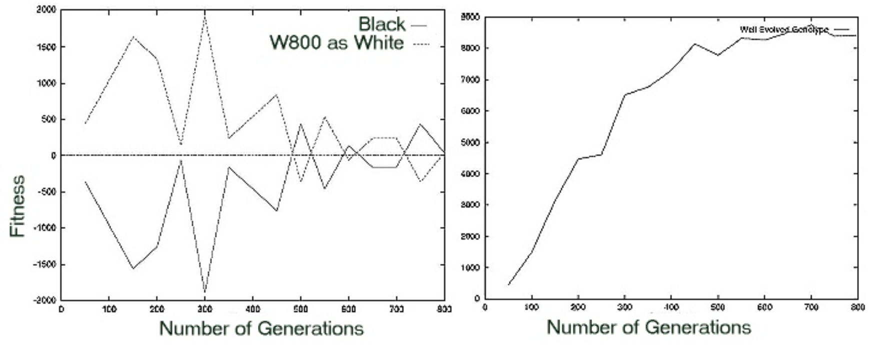


Fig. 3. Graph showing the fitness variation of a well evolved agent playing checker against different generations of less evolved player on right hand, and accomulated fitness variation of high evolved agent on right

Results and Analysis

To learn how to play checkers the agents must start with a few neurons with a random number of dendrites and branches and 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 previous moves and whether they were beneficial or deleterious. Also it should understand the benefits of making a king or jumping over and finally and most importantly it must do all these things while playing the game. Over the generations the agents learn from each other about favourable moves, this learning is transferred through the *genes* from generation to generation.

To test whether more evolved agents play the game better we tested well evolved agents against less evolved agents. We found that the well evolved agent almost always beats 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. Figure 3 shows the variation in fitness of a well evolved agent(from 800 generation) against the fitness of a series of agents from lower generations. From the figure it is evident that the well evolved agent playing white always beat the less evolved playing black. From the cumulative fitness graph(in figure 3 right) of the well evolved agent it is clearly evident that, initially it beats the less evolved agent by large margin, and as the evolution progresses, the variation decreases, showing the increase in learning capabilities of the opposing agents over the course of evolution.

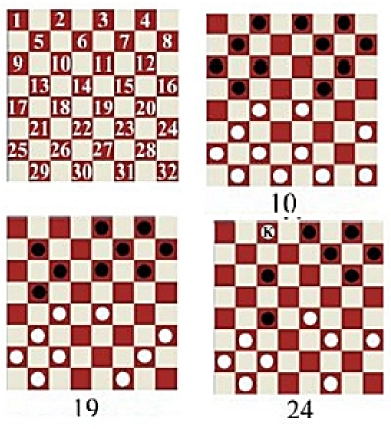
It is instructive to present an annotated game between a highly evolved agent and a less evolved agent. We have numbered the squares on the board as shown in the Figure 4. The highly evolved agent is playing white(from generation 1350) and the less evolved agent(from generation 5) is black.

When the game starts the initial board position is fed into the CGPCN of the agent playing black, the CGPCN is run for five cycles. After this the CGPCN makes a decision about which piece to move and where. The sequence of moves up to move 31 is shown in the Table on the right in figure 4.

Initially both players appear to play sensibly. On move 10, white offers the black the opportunity of taking a piece in two ways. This is a strange move as it was avoidable. However after the ensuing series of forced exchanges, white has its pieces further up the board than black. Move 19 for black is a catastrophic one, as white is forced to take two pieces and acquires a king. Move 24 for black is also disasterous as it results in the forced capture by white of two more of its pieces. It is interesting to observe that both players have learned how to defend pieces by placing pieces behind them. These occurred at moves, 4, 7, 15 and 16. In addition move 31 by black is interesting as moving pieces to edges is a safe thing to do. The game ends up with white winining within 48 moves with one king and eight pieces left.

5 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



Move	Comment	Move	Comment
B1 10 - 13	opening	W2 23 - 19	
B3 11 - 15		W4 28 - 23	defend
B5 5 - 10		W6 23 - 20	
B7 1 - 5	defend but ...	W8 20 - 11	takes
B9 6 - 15	takes	W10 22 - 18	offer
B11 15 - 22	takes	W12 26 - 19	takes
B13 13 - 22	takes	W14 27 - 18	takes
B15 2 - 6	defend	W16 30 - 26	defend
B17 6 - 11		W18 32 - 28	
B19 9 - 13	blunder		
W20 18 - 9	takes	W21 9 - 2	takes, gets K
B22 11 - 14		W23 28 - 23	
B24 14 - 18	blunder		
W25 21 - 14	takes	W26 14 - 5	takes
B27 7 - 11		W28 31 - 27	
B29 11 - 14	blunder	W30 19 - 10	takes
B31 12 - 16	move to edge		

Fig. 4. Labelled Board and positions at different stages of the game. Numbers beneath boards show the board at moves 10, 19 and 24. Table on left lists all the moves played by the two players

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.

References

1. B. Graham. Multiple forms of activity-dependent plasticity enhance information transfer at a dynamic synapse. In *Proc. International Conference on Artificial Neural Networks*, pages 45–50, 2002.
2. G. Khan, J. Miller, and D. Halliday. Coevolution of intelligent agents using cartesian genetic programming. In *Proc. GECCO*, pages 269 – 276, 2007.
3. 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.
4. J. F. Miller and P. Thomson. Cartesian genetic programming. In *Proc. EuroGP*, volume 1802 of *LNCS*, pages 121–132, 2000.
5. J. F. Miller, P. Thomson, and T. C. Fogarty. Designing electronic circuits using evolutionary algorithms. arithmetic circuits: a case study. pages 105–131, 1997.
6. C. Panchev, S. Wermter, and H. Chen. Spike-timing dependent competitive learning of integrate-and-fire neurons with active dendrites. In *Proceedings ICANN*, pages 896–901, 2002.
7. K. Stanley and R. Miikkulainen. Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, 21:63–100, 2004.