On Dynamical Genetic Programming:

Simple Boolean Networks in Learning Classifier Systems

Larry Bull

Learning Classifier Systems Group Technical Report – *UWELCSG08-006*

University of the West of England

Bristol BS16 1QY, U.K.

Larry.Bull@uwe.ac.uk

Oct. 2008

Abstract. Many representations have been presented to enable the effective evolution of computer programs. Turing was perhaps the first to present a general scheme by which to achieve this end. Significantly, Turing proposed a form of discrete dynamical system and yet dynamical representations remain almost unexplored within conventional genetic programming. This paper presents results from an initial investigation into using simple dynamical genetic programming representations within a Learning Classifier System. It is shown possible to evolve ensembles of dynamical Boolean function networks to solve versions of the well-known multiplexer problem. Both synchronous and asynchronous systems are considered.

Keywords: discrete, dynamical systems, evolution, multiplexer, unorganised machines.

INTRODUCTION

In 1948 Alan Turing produced a paper entitled “Intelligent Machinery” in which he highlighted evolutionary search as a possible means by which to program machines (e.g., see [Copeland, 2004] for an overview). In the same paper, Turing also presented a formalism he termed “unorganised machines” by which to represent intelligence within computers. These consisted of two types: A-type unorganised machines, which were composed of two-input NAND gates randomly connected into networks (Figure 1); and, B-type unorganised machines which included an extra triplet of NAND gates on the arcs between the NAND gates of A-type machines by which to affect their behaviour in a supervised learning-like scheme through the constant application of appropriate extra inputs to the network (Figure 2). In both cases, each NAND gate node updates in parallel on a discrete time step with the output from each node arriving at the input of the node(s) on each connection for the next time step. The structure of unorganised machines is very much like an artificial neural network with recurrent connections and hence it is perhaps surprising that Turing made no reference to McCulloch and Pitts’ [1943] prior seminal paper on networks of binary-thresholded nodes. However, Turing’s scheme extended McCulloch and Pitts’ work in that he also considered the training of such networks with his B-type architecture. This has led to their also being known as “Turing’s connectionism” [Copeland & Proudfoot, 1996]. Moreover, as Teuscher [2002] has highlighted, Turing’s unorganised machines are (discrete) nonlinear dynamical systems and therefore have the potential to exhibit complex behaviour despite their construction from simple elements. Around the same time as Turing was working on artificial intelligence in the 1940’s, John von Neumann, together with Stanislaw Ulam, developed the regular lattice-based discrete dynamical systems known as Cellular Automata (CA) [von Neumann, 1966] to explore self-reproduction. That is, CAs are (traditionally) discrete dynamical systems which exist on a graph of restricted connectivity but with potentially any logical function at each node, whereas unorganised machines exist on a graph of potentially any connectivity topology but with a restricted logical function at each node. The current work aims to exploit the potential for emergent computation of such graph-based systems through the use of evolutionary search, what is herein termed “dynamical genetic programming” (DGP). In particular, node asynchrony and open-ended, self-adaptive simulated evolution is utilized to design ensembles. Since B-type unorganised machines were introduced as a form of supervised learner they are not considered here – only A-type machines are explored.

Figure 1: A-type unorganised machine consisting of four two-input NAND gates.

Figure 2: B-type unorganised machine consisting of four two-input NAND gates. Each connecting arc contains a three NAND gate “interference” mechanism so that external inputs such as S1 and S2 can be applied to affect overall behaviour, i.e., a form of supervised learning.

EVOLVING DISCRETE DYNAMICAL SYSTEMS

The most common form of discrete dynamical system is the Cellular Automaton which consists of an array of cells where the cells exist in states from a finite set and update their states in parallel in discrete time. Traditionally, each cell calculates its next state depending upon its current state and the states of its closest neighbours. Packard [1988] was the first to use evolutionary computing techniques to design CAs such that they exhibit a given emergent global behaviour. Following Packard, Mitchell et al. (e.g., [1993]) have investigated the use of a Genetic Algorithm (GA) [Holland, 1975] to learn the rules of uniform one-dimensional, binary CAs. As in Packard's work, the GA produces the entries in the update table used by each cell, candidate solutions being evaluated with regard to their degree of success for the given task — density and synchronization. Sapin et al. (e.g., [2003]) have used the approach to discover novel moving structures - “gliders” - within two-dimensional CAs. Andre et al. [1999] repeated Mitchell et al.'s work evolving the tree-based LISP S-expressions of Genetic Programming (GP) [Koza, 1992] to identify the update rules. They report similar results. Sipper (e.g., [1997]) presented a non-uniform, or heterogeneous, approach to evolving CAs. Each cell of a one- or two-dimensional CA is also viewed as a GA population member, mating only with its lattice neighbours and receiving an individual fitness. He shows an increase in performance over Mitchell et al.'s work by exploiting the potential for spatial heterogeneity in the tasks. More recently, a variation of this approach has been used to construct simple chemical logic gates within a non-linear reaction (e.g., [Toth et al., 2008]). Hence for twenty years it has been known that it is possible to design emergent computation from a graph-based discrete dynamical system through simulated evolution. However, the types of computation produced in this way have been less than general in the traditional programming sense. GP represents a way in which to use simulated evolution to design programs in general and some synchronous dynamical programs have previously been evolved.

EVOLVING GRAPH-BASED PROGRAMS

A number of representations have been presented by which to enable the evolution of computer programs, the most common being tree-based representations (e.g., [Cramer, 1985][Fujiki & Dickinson, 1987][Koza, 1989]). Other forms of Genetic Programming use the more traditional linear representation, e.g., of machine code instructions (e.g., [Friedberg, 1958][Ray, 1992][Banzhaf, 1993]) or Boolean

expressions [Forsyth, 1981]. Most relevant to the form of GP to be explored in this paper is the relatively small amount of prior work on graph-based representations. Fogel et al. (e.g., [1965]) were the first to evolve graph-based (sequential) programs with their use of finite state machines. The connection structure - as opposed to the connection weights - of artificial neural networks was first evolved by Dolan and Dyer (e.g., [1987]). Teller and Veloso's [1996] "neural programming" (NP) uses a directed graph of connected nodes, each with functionality defined in the standard GP way, with recursive connections included. Here each node executes in synchronous parallelism for some number of cycles before an output node's value is taken. Luke and Spector [1996] presented an indirect, or cellular, encoding scheme by which to produce graphs, as had been used to design artificial neural networks (e.g., [Gruau & Whitley, 1993]), an approach recently used to design both unorganised machines [Teuscher, 2002] and automata networks with synchronously parallel nodes [Leurssen & Powers, 2005]. Poli [1997] presented a very general scheme wherein nodes are connected in a graph which is placed over a two-dimensional grid. Later, recurrent artificial neural networks were designed such that the nodes were synchronously parallel (e.g., [Pujol & Poli, 1998]) and variants exist in which some nodes can update more frequently than others (see [Poli, 1999] for an overview). Miller (e.g., [1999]) has presented a graph-based representation scheme originally designed to consider the hardware implementation of the evolved program wherein a two-dimensional grid of sequentially updating, connected logic blocks is produced. Other examples of sequentially updating nodes include [Niehaus & Banzhaf, 2001] and [Hiraswa et al., 2001]. Schmidt and Lipson [2007] have recently demonstrated a number of benefits from graph encodings in general over traditional trees, such as reduced bloat and increased computational efficiency, and Galvan-Lopez [2008] has highlighted how multiple, simultaneous outputs are possible from graphs.

The motivating idea behind this work is that Turing's initial scheme can be augmented with elements of more recent discrete dynamical systems research and evolutionary computing to create a flexible and robust approach to the automated design of emergent computation for difficult problems. That is, it is proposed that using self-adaptive simulated evolution to shape asynchronous computer programs capable of rich temporal behaviour in themselves will enable the effective control or prediction of systems which contain complex dynamics. It can also be noted that dynamical systems represent a way in which to view cognition in general (e.g., [Ashby, 1952][Port & van Gelder, 1995]).

EVOLVING UNORGANISED MACHINES

A-type unorganised machines have a finite number of possible states and they are deterministic, hence such networks eventually fall into a basin of attraction. Turing [1948] was aware that his A-type unorganised machines would have periodic behaviour and he stated that since they represent “about the simplest model of a nervous system with a random arrangement of neurons” it would be “of very great interest to find out something about their behaviour” [ibid.]. Figure 3 shows the fraction of nodes which change state per update cycle for 100 randomly created networks, each started from a random initial configuration, for various numbers of nodes N . As can be seen, the time taken to equilibrium is typically around 10 cycles, with all nodes changing state on each cycle thereafter, i.e., oscillating. For the smaller networks ($N=5$, $N=50$), some nodes remain unchanging at equilibrium however; with smaller networks, the probability of nodes being isolated is sufficient that the basin of attraction contains a degree of node stasis (see [Teuscher, 2002] for a similar study).

Figure 3: Showing the average fraction of two-input NAND gate nodes which change state per update cycle of random A-type unorganised machines with various numbers of nodes N .

Previously, Teuscher [2002] has explored the use of evolutionary computing to design both A-type and B-type unorganised machines together with new variants of the latter. In his simplest encoding, an A-type machine is represented by a string of N pairs of integers, each integer representing the node number within the network from which that NAND gate node receives an input. Turing [1948] did not explicitly demonstrate how inputs and outputs were to be determined for A-type unorganised machines. Teuscher [2002] used I input nodes for I possible inputs, each of which receive the external input only and are then connected to any of the nodes within the network as usual connections. He then allows for O outputs from anywhere within the network. Thus his scheme departs slightly from Turing's for B-type unorganised machines since Turing there showed the input nodes receiving the external input and an input from another node within the network (Figure 2). Teuscher uses his own scheme for all of his work on unorganised machines, which may be viewed as directly analogous to specifying the source of inputs via a terminal set in traditional tree-based GP. The significance of this difference is not explored here and Turing's input scheme is used throughout.

Teuscher used a GA to design single A-type unorganised machines for bitstream regeneration tasks and simple pattern classification. In the former case, the size of the networks, i.e., the number of nodes, was increased by one after every 30,000 generations until a solution was found. That is, an epochal approach was exploited to tackle the issue of not knowing how complex an A-type unorganised machine will need to be for a given task. In this paper, parameter self-adaptation is used to enable the solution size to emerge during the course of evolution via a mutation operator. Further, a Learning Classifier System (LCS) [Holland, 1976] is used as the underlying architecture. In particular, LCS evolve an ensemble of solutions to a given task (see [Drugowitsch, 2008] for discussions) wherein divisions of the problem space emerge along with their solution. To date, no temporally dynamic representation scheme has been used within LCS. A number of representations have previously been presented beyond the traditional binary scheme however, including integers [Wilson, 2001], real numbers [Wilson, 2000], Lisp S-expressions (e.g., [Ahluwalia & Bull, 1999][Lanzi & Perrucci, 1999]), fuzzy logic [Venzuela-Rendon, 1991] and neural networks [Bull, 2002]. Thus this paper also represents an initial study into the use of simple forms of dynamical system within LCS. Hereafter A-type unorganised machines will be referred to by the more descriptive label random NAND networks (RNN). Following Turing and all of the above mentioned

previous research, synchronous node updating is first considered together with the underlying evolutionary design process.

DYNAMICAL GP IN A LEARNING CLASSIFIER SYSTEM

In this paper a version of the simple accuracy-based LCS termed YCS [Bull, 2005a] - which is a derivative of Wilson's XCS [Wilson, 1995] - is used. YCS is without internal memory and maintains a rulebase of P initially randomly created rules. Associated with each rule is a predicted payoff value (p), a scalar which indicates the error (ε) in the rule's predicted payoff and an estimate of the average size of the niches (action sets - see below) in which that rule participates (σ). The initial random population has these parameters initialized, somewhat arbitrarily, to 10.

On receipt of an input message, the rulebase is scanned, and any rule whose condition matches the message at each position is tagged as a member of the current match set $[M]$. An action is then chosen from those proposed by the members of the match set and all rules proposing the selected action form an action set $[A]$. A version of XCS's explore/exploit action selection scheme will be used here. That is, on one cycle an action is chosen at random and on the following the action with the highest average payoff is chosen deterministically. No learning occurs on exploit trials – they are simply used to indicate progress.

The simplest case of immediate payoff reward R is considered here. Reinforcement in YCS consists of updating the error, the niche size estimate and then the payoff estimate of each member of the current $[A]$ using the Widrow-Hoff delta rule with learning rate β :

$$\varepsilon_j \leftarrow \varepsilon_j + \beta(|R - p_j| - \varepsilon_j) \quad (1)$$

$$\sigma_j \leftarrow \sigma_j + \beta(|[A]| - \sigma_j) \quad (2)$$

$$p_j \leftarrow p_j + \beta(R - p_j) \quad (3)$$

The original YCS employs two discovery mechanisms, a panmictic (standard global) GA and a covering operator. On each time-step there is a probability g of GA invocation. The GA uses roulette wheel selection to determine two parent rules based on the inverse of their error:

$$f_j = (1 / (\epsilon_j^\nu + 1)) \quad (4)$$

Here the exponent ν enables control of the fitness pressure within the system by facilitating tuneable fitness separation under fitness proportionate selection (see [Bull, 2005a] for discussions). Offspring are produced via mutation (probability μ) and crossover (single point with probability χ), inheriting the parents' parameter values or their average if crossover is invoked. Replacement of existing members of the rulebase uses roulette wheel selection based on estimated niche size. If no rules match on a given time step, then a covering operator is used which creates a rule with the message as its condition and a random action, which then replaces an existing member of the rulebase in the usual way. Parameter updating and the GA are not used on exploit trials.

In this paper, to aid the generalization process, the panmictic GA is altered to operate within niches (e.g., see [Butz et al., 2004][Bull, 2005a] for discussions). The mechanism uses XCS's time-based approach under which each rule maintains a time-stamp of the last system cycle upon which it was part of a GA. The GA is applied within the current [A] when the average number of system cycles since the last GA in the set is over a threshold θ_{GA} . If this condition is met, the GA time-stamp of each rule is set to the current system time, two parents are chosen according to their fitness using standard roulette-wheel selection, and their offspring are potentially crossed and mutated, before being inserted into the rulebase as described above.

To use RNN as the rules within this system the following scheme is adopted. Each of an initial randomly created rule's nodes has 2 randomly assigned connections. There are initially as many nodes N_{init} as input fields I for the given task and its outputs O , plus one other, as will be described, i.e., $N_{init}=I+O+1$. The

first connection of each input node is set to the corresponding locus of the input message. The other connection is assigned at random within the RNN as usual. In this way the current input state is always considered along with the current state of the RNN itself per network update cycle by such nodes.

Matching consists of executing each rule for T cycles based on the current input. The value of T is typically chosen to be well within the basin of attraction of the RNN. Nodes are initialised randomly. In this initial study well-known Boolean problems are explored and hence there are only two possible actions, meaning only one output node is required. An extra “matching” node is also required to enable RNNs to (potentially) only match specific sets of input. If a given RNN has a logical ‘1’ on the match node, regardless of its output node’s state, the rule does not join [M]. This scheme has also been exploited within neural LCS [Bull, 2002]. Thereafter match set and action set processing proceeds as described above. A cover operator has not been found necessary in the tasks explored here.

Due to the need for a possible different number of nodes within the rules for a given task, the representation scheme is of variable length. In this initial study, mutation only is used and applied to the connectivity map at rate μ ; as in Teuscher’s [2002] representation, the inputs of each node are essentially concatenated together to form a string of length $2N_i$. Typically in LCS, as within GAs, the parameters controlling the algorithm are global and remain constant over time. However this is not always the case in evolutionary computing; in Evolution Strategies (ES) [Schwefel, 1981], forms of Evolutionary Programming (Meta-EP) [Fogel et al., 1991] and in some GAs (e.g., [Bäck, 1992]), the mutation rate is a locally evolving entity in itself, i.e., it adapts during the search process. The first use of self-adaptive mutation was by Reed et al. [1967]. Self-adaptive mutation not only reduces the number of hand-tunable parameters of the evolutionary algorithm, it has also been shown to improve performance (e.g., [Bäck, 1992]). The approach has been used to add adaptive mutation to LCS [Bull et al., 2000], and to control other system parameters, such as the learning rate (e.g., [Hurst & Bull, 2001]). The results demonstrate that adaptive parameters can provide improved performance, particularly in dynamic environments. Here each rule has its own mutation rate μ , stored as a real number and initially seeded uniform randomly in the range [0.0,1.0]. This parameter is passed to its offspring. The offspring then applies its mutation rate to itself using a Gaussian distribution, i.e., $\mu' = \mu e^{N(0,1)}$, before mutating the rest of the rule at the resulting rate.

As noted above, this process is also used to enable the number of nodes, i.e., the complexity of the RNN, to vary to match the task. Each rule also contains a second mutation rate γ , adjusted in the same way as μ . Once standard mutation is applied, the probability γ is tested. Upon satisfaction, a new randomly connected node (not to inputs) is either added or the last added node is removed. The latter case only occurs if the network currently consists of more than the initial number of nodes. This self-adaptive growth scheme has previously been used within neural LCS (e.g., [Hurst & Bull, 2006]), within GAs [Bull, 2005b], and ES [Bull, 2008]. Evolving variable-length solutions via mutation only has previously been explored a number of times (e.g., [Fogel et al., 1965][Harvey, 1992]). Whereas traditional GP can be seen to primarily rely upon recombination to search the space of possible tree sizes, although the standard mutation operator effectively increases or decreases tree size also.

All other GA processing of the LCS is as described above.

Figure 4: Example rule for a 3-input problem, with two extra nodes added from the typical initialisation state (nodes labelled 4 and 5 here).

EXPERIMENTATION

Versions of the well-known multiplexer task are used in this paper. These Boolean functions are defined for binary strings of length $l = x + 2^x$ under which the x bits index into the remaining 2^x bits, returning the value of the indexed bit. The correct classification of a randomly generated instance results in a payoff of 1000, otherwise 0.

Figure 5 shows the performance of the RNN-LCS on the 6-bit multiplexer problem with most parameters taken from [Bull, 2005a]: $P=2000$, $v=10$, $\theta_{GA}=25$, $\beta=0.2$, $T=100$ and $N_{init} = 8$ (6 inputs, one output, one match node). After [Wilson, 1995], performance from exploit trials only is recorded (fraction of correct responses are shown), using a 50-point running average, averaged over ten runs. It can be seen that optimal performance is not reached and performance levels at around 95% after 200,000 trials, with the average error dropping to roughly 10% of the payoff range. The mutation and growth rates adapt differently, with a significant level of network size tuning being maintained in contrast to the standard monotonically decreasing mutation rate. The continued exploration of network size is apparent in the lower graph in Figure 5 where a slow but steady increase in average size is seen.

Figure 5: Showing the behaviour of two-input NAND gate nodes on the 6-MUX.

In efforts to improve performance, Turing's [1948] restriction of only 2 inputs per node was relaxed to enable evolution to more fully explore the space of possible programs. Each initial randomly created rule has a randomly assigned degree of connectivity K , here $1 \leq K \leq 5$. Again, the first connection of each input node is set to the corresponding locus of the input message and now the other $K-1$ connections are assigned at random within the RNN as usual. Examination of the underlying dynamics of RNN with varying K indicates that for a given N , the time taken to reach a basin of attraction decreases with increasing K . Figure 6 shows the case for $N=5000$ (compare to Figure 3). The number of isolated nodes which remain unchanging at equilibrium for low N decreases with increasing K also (not shown).

Figure 6: Showing the average fraction of two-input NAND gate nodes which change state per update cycle of random A-type unorganised machines with various K and $N=5000$.

Figure 7 shows a significant decrease in the time taken to reach maximal performance under this scheme but the final performance is the same at around 95%. The average error and mutation rates are very similar to those in Figure 5 but there is a slight decrease in the sustained level of network size mutation, along with a corresponding reduction in the average size of networks. It is interesting to note that the average connectivity converges to around 2, i.e., as in Turing's original scheme, but the variability in connectivity obviously aids evolution. This point is returned to in the next Section.

In the aforementioned work on evolving CAs, the global behaviour of the lattice is typically used to determine the output of the discrete dynamical system. The same approach has also been explored here such that if a fraction of nodes ϕ within the given rule are in state '0', the rule is said to match the current input and advocate action '0'. Conversely, if the fraction ϕ of nodes are in state '1', the rule is said to match and advocate action '1'. Otherwise the rule is deemed not to match the current input. Under this scheme $N_{init} = I$. $\phi = 0.75$ is used here. However, the results in Figure 8 suggest that this is less beneficial to the evolutionary process with performance dropping to around 70%. Generally the same results are seen when using the K -input gate scheme described above although it gives a slight improvement in performance to around 80% (not shown).

Figure 7: Showing the behaviour of K -input NAND gate nodes on the 6-MUX.

Figure 8: Showing the behaviour of two-input NAND gate nodes on the 6-MUX where a percentage of global network state is used to determine the output.

RNN can be viewed as a form of disordered CA since they are homogeneous in their node function but heterogeneous in their connectivity. As noted above, Sipper [1997] has reported improved performance over the evolution of standard CAs on a number of well-known tasks by enabling the potential for heterogeneity in the node function whilst maintaining homogeneity in connectivity. That is, Sipper's results suggest that node homogeneity can reduce evolvability in discrete dynamical systems (see [Wagner & Altenberg, 1996] for general discussions of evolvability). The use of heterogeneity in the nodes is now explored.

RANDOM BOOLEAN NETWORKS

The discrete dynamical systems known as random Boolean networks (RBN) were originally introduced by Kauffman [1969] to explore aspects of biological genetic regulatory networks. Since then they have been used as a tool in a wide range of areas such as self-organisation [e.g., Kauffman, 1993], computation [e.g., Mesot & Teuscher, 2005] and robotics [e.g., Quick et al., 2003]. An RBN typically consists of a network of N nodes, each performing a Boolean function with K inputs from other nodes in the network, all updating synchronously. As such, RBN may be viewed as a generalization of Turing's A-type unorganised machines which used only the NAND Boolean function with $K=2$. Turing's paper was not published until 1968 [Turing, 1968] so it is perhaps not too surprising that Kauffman did not originally discuss his work - although no connection has been made subsequently either, except in [Teuscher, 2002].

It is well-established that the value of K affects the emergent behaviour of RBN wherein attractors typically contain an increasing number of states with increasing K . Three phases of behaviour are suggested: ordered when $K=1$, with attractors consisting of one or a few states; chaotic when $K>3$, with a very large numbers of states per attractor; and, a critical regime around $K=2$, where similar states lie on trajectories that tend to neither diverge nor converge and 5-15% of nodes change state per attractor cycle (see [Kauffman, 1993] for discussions of this critical regime, e.g., with respect to perturbations). Analytical methods have been presented by which to determine the typical time taken to reach a basin of attraction and the number of states within such basins for a given degree of connectivity K [e.g., Kauffman, 1993].

A very small number of studies have considered multiple, coupled RBN (e.g., [Hung et al., 2006][Bull & Alonso-Sanz, 2008]) but no previous consideration of ensemble scenarios, as here, are known.

Previously, Van den Broeck and Kawai [1990] explored the use of a simulated annealing-type approach to design feedforward RBN for the four-bit parity problem. Lemke et al. [2001] have evolved fixed RBN of fixed N and K to match an arbitrary attractor. More closely akin to the work described here, Kauffman [1993, p.223] describes the use of simulated evolution to design RBN which must play a (mis)matching game wherein mutation is used to change connectivity, the Boolean functions, K and N . He reports the typical emergence of high fitness solutions with $K=2$ to 3, together with an increase in N over the

initialised size which “suggests that, typically, larger, more complex networks can solve the mismatch task better” [ibid.].

The representation scheme described above for RNN is here augmented with a binary string associated with each node by which its Boolean logic function is defined. Since each rule has a given K value, each node maintains a binary string of length 2^K which forms the entries in the look-up table for each of the possible 2^K input states of that node, i.e., as in Packard’s [1988] aforementioned work on evolving CAs, for example. These strings are subjected to mutation on reproduction at the self-adapting rate μ for that rule. Hence, within the RBN representation, evolution can define different Boolean functions for each node within a given network rule, along with its connectivity map.

RBN EXPERIMENTATION

Figure 9 shows the performance of the RBN-LCS on the 6-bit multiplexer problem with same parameters and set-up as in Figure 4: $P=2000$, $v=10$, $\theta_{GA}=25$, $\beta=0.2$, $T=100$ and $N_{init} = 8$ (6 inputs, one output, one match node). It can be seen that a near optimal solution is learnt around 30,000 trials with optimality seen around trial 70,000. The average degree of connectivity K converges to around 2, i.e., connectivity evolves to the aforementioned critical regime identified for RBN in general. This behaviour was also seen with the RNN representation in Figure 6 and indicates that the evolutionary process is able to identify an appropriate typical topology with which to generate complex behaviour, i.e., in this case a computation. For other tasks, other values of K may prove beneficial; high K may be expected in random number generation, for example. The average error and mutation rates drop quickly, with the growth mutation dropping more slowly and a corresponding slight amount of growth is seen. Figure 10 shows an example rule from early in learning which is correct for the input “01” on the address bits where a direct correspondence between the connectivity of data line 3, i.e., input 5, at node 4 and the action node 6 can be seen. The Boolean function of node 6 sets the node to the state of node 4. The rule always matches and so has high error in general, but better than average/random due to its being correct in these cases.

Figure 11 shows the performance of the system when the network-level output is used, as in Figure 8. It can be seen that optimal performance is reached around 20,000 trials, with the average error dropping to almost zero, along with the mutation rate. Therefore this “population encoding” scheme appears more

robust with the RBN representation. Some size mutation is maintained but the average number of nodes remains around 6 after an initial growth period.

Figure 9: Showing the behaviour of the RBN representation on the 6-MUX task.

Figure 10: Showing an example rule with the RBN representation on the 6-MUX task which is correct for the input “01” on the address bits early in learning. Analysis of the nodes’ functions finds the action node 6 is connected to node 4 with a logic that simply turns it to state one if node four is in state one, otherwise it remains at zero, and the matching node is always set to state zero, i.e., matching. Decoding rules becomes more complex later in learning.

Figure 11: Showing the behaviour of the RBN representation on the 6-MUX where a percentage of global network state is used to determine the output.

Thus in both cases, the RBN representation has enabled the system to perform optimally which is a significant improvement over the pure NAND gate representation of Turing's scheme for the given parameters and general set-up used here; node function heterogeneity appears to aid evolvability, as suggested by Sipper's [1997] previous results with CAs.

Figure 12 shows the performance of the system with the same parameters on the 11-bit multiplexer using the network level output scheme, with optimal performance realised around 100,000 learning trials. The average error drops to around 5% of the payoff range, the average connectivity of rules is approximately 1.75 and the networks typically grow one extra node. Similar results were found for the output node configuration, with it again being slower to optimality than the network version (not shown).

As a rough benchmark, whilst direct comparison is perhaps difficult due to the use of a different training scheme, it can be noted that Koza [1992] reports $\sim 7.3 \times 10^7$ evaluations being required with traditional GP to solve the 11-bit problem, i.e., considerably more than the $\sim 1 \times 10^5$ evaluations seen in Figure 12.

Figure 12: Showing the behaviour of the RBN representation on the 11-MUX where a percentage of global network state is used to determine the output.

ASYNCHRONOUS DYNAMICAL GENETIC PROGRAMMING IN LCS

As noted above, traditional RBN consist of N nodes updating synchronously in discrete time steps but asynchronous versions have also been presented, after [Harvey & Bossomaier, 1997], leading to a classification of the space of possible forms of RBN [Gershenson, 2002]. Asynchronous forms of CA have also been explored (e.g., [Nakamura, 1974][Ingerson & Buvel, 1984][Bersini & Detour, 1994]) wherein it is often suggested that asynchrony is a more realistic underlying assumption for many natural and artificial systems. Asynchronous logic devices are also known to have the potential to consume less power and dissipate less heat [Werner & Akella, 1997], which may be exploitable during efforts towards hardware implementations of such systems (e.g., see [Ferrandi et al., 2004] for a hardware LCS). Asynchronous logic is also known to have the potential for improved fault tolerance, particularly through delay insensitive schemes (e.g., [Di & Lala, 2007]). This may also prove beneficial for direct hardware implementations of DGP (e.g., as an extension to the aforementioned work in [Miller, 1999]). Harvey and Bossomaier [1997] showed that asynchronous RBN exhibit either point attractors, as seen in asynchronous CAs, or “loose” attractors where “the network passes indefinitely through a subset of its possible states” [ibid.] (as opposed to distinct cycles in the synchronous case). Thus the use of asynchrony represents another feature of dynamical systems with the potential to significantly alter their underlying behaviour thereby offering another mechanism by which to aid the simulated evolutionary design process for a given task. For these reasons, asynchronous node updating is hereafter considered the default for DGP.

Di Paolo [2001] showed it is possible to evolve asynchronous RBN which exhibit rhythmic behaviour at equilibrium. Asynchronous CAs have also been evolved (e.g., [Sipper et al., 1997][Tomassini & Venzi, 2002]). No prior work on the use of asynchronous RBN for computation is known and neither is any prior work on asynchronous node/instruction updating in any other form of GP.

Asynchrony is here implemented as a randomly chosen node being updated on a given cycle, with as many updates per overall network update cycle as there are nodes in the network before an equivalent cycle to one in the synchronous case is said to have occurred (see [Gershenson, 2002] for alternative schemes).

Figure 13 shows how, despite their potentially very different underlying dynamics to synchronous RBN, the performance of the two systems is very similar (compare with Figure 12). The only marked difference is in the evolution of smaller networks on average; initial growth is typically lost in the asynchronous case. Similar results are found for the node output scheme (not shown).

Figure 13: Showing the behaviour of the asynchronous RBN representation on the 11-MUX where a percentage of global network state is used to determine the output.

CONCLUSIONS

Sixty years after Turing's seminal work and almost forty years after Kauffman's presentation of RBN, this paper has explored discrete Boolean forms of dynamical GP. In particular, a form of LCS has been presented with which to design ensembles of dynamical genetic programs. It has here been shown that evolutionary search is able to design ensembles that collectively solve a computational task under the reinforcement learning scheme of LCS. It can be noted that Forrest and Miller [1989] used RBN to model the internal rule chaining of traditional LCS.

It has been shown that Turing's purely NAND-based graph representation scheme appears difficult for the simulated evolutionary approach employed to shape for the chosen tasks. Rather, node heterogeneity in the form of RBN, greatly improves performance. This finding concurs with those of Sipper [1997] on the evolution of another form of discrete dynamical system, CAs. It has also been shown that node function, degree of connectivity, connectivity map, and number can be left to the control of self-adaptive mutation processes. These findings suggest that dynamical GP has significant potential as a general approach to the construction of asynchronous programs.

Current research is exploring many of the myriad of possibilities dynamical GP presents as a form of emergent computation by which to solve complex problems. Most importantly, it is being applied to tasks where the evolution of their temporal behaviour can be exploited directly, i.e., online, as ensembles within LCS, such as time-series data mining and adaptive control.

REFERENCES

Ahluwalia, M. & Bull, L. (1999) A Genetic Programming Classifier System. In W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela & R.E. Smith (Eds) *Proceedings of the Genetic and Evolutionary Computation Conference – GECCO-99*. Morgan Kaufmann, pp11-18.

Andre, D., Koza, J.R., Bennett, F.H. & Keane, M. (1999) *Genetic Programming III*. MIT Press.

Ashby, W. R. (1952) *Design for a Brain*. Chapman & Hall.

Back, T. (1992) Self-Adaptation in Genetic Algorithms. In F. Varela & P. Bourguine (Eds) *Proceedings of 1st European Conference on Artificial Life*. MIT Press, pp263-271.

Banzhaf, W. (1993) Genetic Programming for Pedestrians. In S. Forrest (Ed) *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufmann, pp628.

Bersini, H. & Detours, V. (1994) Asynchrony Induces Stability in Cellular Automata based Models. In R.A. Brooks & P. Maes (Eds) *Artificial Life IV*. MIT Press, pp382-387.

Bull, L. (2002) On Using Constructivism in Neural Classifier Systems. In J. Merelo, P. Adamidis, H-G. Beyer, J-L. Fernandez-Villacanas & H-P. Schwefel (Eds) *Parallel Problem Solving from Nature - PPSN VII*. Springer Verlag, pp558-567.

Bull, L. (2005a) Two Simple Learning Classifier Systems. In L. Bull & T. Kovacs (eds) *Foundations of Learning Classifier Systems*. Springer, pp63-90.

Bull, L. (2005b) Coevolutionary Species Adaptation Genetic Algorithms: A Continuing SAGA on Coupled Fitness Landscapes. In M. Capcarrere et al. (Eds) *Proceedings of the Eighth European Conference on Artificial Life*. Springer, pp322-331.

Bull, L. (2008) Toward Artificial Creativity with Evolution Strategies. In I. Parmee (Ed) *Adaptive Computing in Design and Manufacture VIII*. IPCC.

Bull, L. Hurst, J. & Tomlinson, A. (2000) Self-Adaptive Mutation in Classifier System Controllers. In J. A. Meyer, A. Berthoz, D. Floreano, H. Roitblatt & S.W. Wilson (Eds) *From Animals to Animats 6*. MIT Press, pp460-468.

Bull, L. & Alonso-Sanz, A. (2008) On Coupling Random Boolean Networks. In A. Adamatzky, R. Alonso-Sanz, A. Lawniczak, G. Juarez Martinez, K. Morita & T. Worsch (Eds) *Automata 2008: Theory and Applications of Cellular Automata*. Luniver Press, pp292-301.

Butz, M., Kovacs, T., Lanzi, P.L. & Wilson, S.W. (2004) Toward a Theory of Generalization and Learning in XCS. *IEEE Transactions on Evolutionary Computation* 8(1): 28-46.

Cramer, N. (1985) A Representation for the Adaptive Generation of Simple Sequential Programs. In J.J. Grefenstette (Ed) *Proceedings of an International Conference on Genetic Algorithms and their Application*. Lawrence Erlbaum, pp183-187.

Copeland, J. (2004) *The Essential Turing*. Oxford.

Copeland, J. & Proudfoot, D. (1996) On Alan Turing's Anticipation of Connectionism. *Synthese* 108: 361-377.

Di, J. & Lala, P. (2007) Cellular Array-based Delay Insensitive Asynchronous Circuits Design and Test for Nanocomputing Systems. *Journal of Electronic Testing* 23: 175-192.

Di Paolo, E. A. (2001) Rhythmic and Non-rhythmic Attractors in Asynchronous Random Boolean Networks. *Biosystems*, 59(3):185–195.

Dolan, C. & Dyer, M. (1987) Toward the Evolution of Symbols. In J.J. Grefenstette (Ed) *Proceedings of the Second International Conference on Genetic Algorithms*. Lawrence Erlbaum, pp123-131.

Drugowitsch, J. (2008) *Design and Analysis of Learning Classifier Systems*. Springer.

Ferrandi, F., Lanzi, P-L. & Sciuto, D. (2004) System Level Hardware–Software Design Exploration with XCS. In *Proceedings of the Genetic and Evolutionary Computation Conference – GECCO-04*. Springer, pp763-773.

Friedberg, R. M. (1958) A Learning Machine: Part I. *IBM Journal of Research and Development* 2(1): 2-13.

Fogel, D.B., Fogel, L.J. & Atmar, W. (1991) Meta-Evolutionary Programming. In R. Chen (Ed) *Proceedings of the 25th Asilomar Conference on Signals, Systems and Computers*. IEEE Computer Society, pp540-545.

Fogel, L. J., Owens, A.J. & Walsh, M.J. (1965) Artificial Intelligence Through A Simulation of Evolution. In M. Maxfield, A. Callahan & L.J. Fogel (Eds) *Biophysics and Cybernetic Systems: Proceedings of the 2nd Cybernetic Sciences Symposium*. Spartan Books, pp131-155.

Forrest, S. & Miller, J.H. (1989) The Dynamical Behaviour of Classifier Systems. In J.D. Schaffer (Ed) *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann, pp304-310.

Forsyth, R. (1981) BEAGLE: A Darwinian Approach to Pattern Recognition. *Kybernetes* 10: 159-166.

Fujiki, C. & Dickinson, J. (1987) Using the Genetic Algorithm to Generate LISP Source Code to Solve the Prisoner's Dilemma. In J.J. Grefenstette (Ed) *Proceedings of the Second International Conference on Genetic Algorithms*. Lawrence Erlbaum, pp236-240.

Galvan-Lopez, E. (2008) Efficient Graph-based Genetic Programming Representation with Multiple Outputs. *International Journal of Automation and Computing* 5(1).

Gershenson, C. (2002) Classification of Random Boolean Networks. In R.K. Standish, M. Bedau & H. Abbass (Eds.) *Artificial Life VIII*. MIT Press, pp1-8.

Gruau, F. & Whitley, D. (1993) Adding Learning to the Cellular Development Process. *Evolutionary Computation* 1(3): 213-233.

- Harvey, I. (1992) Species Adaptation Genetic Algorithms: A Basis for a Continuing SAGA. In F. Varela & P. Bourguine (Eds) *Proceedings of 1st European Conference on Artificial Life*. MIT Press, pp346-354.
- Harvey, I. & Bossomaier, T. (1997) Time out of Joint: Attractors in Asynchronous Random Boolean Networks. In P. Husbands & I. Harvey (Eds) *Proceedings of the Fourth European Artificial Life Conference*. MIT Press, pp67-75.
- Hirasawa, K., Okubo, M., Katagari, H., Hu, J. & Murata, J. (2001) Comparison between Genetic Network Programming (GNP) and Genetic Programming (GP). In *Proceedings of the 2001 Congress on Evolutionary Computation*. IEEE Press, pp1276-1282.
- Holland, J.H. (1975) *Adaptation in Natural and Artificial Systems*. University of Michigan Press.
- Holland, J.H. (1976) Adaptation. In R. Rosen & F.M. Snell (Eds) *Progress in Theoretical Biology 4*. Plenum, pp263-293.
- Hung, Y-C., Ho, M-C., Lih, J-S. & Jiang, I-M. (2006) Chaos Synchronisation in two Stochastically Coupled Random Boolean Networks. *Physics Letters A* 356: 35-43.
- Hurst, J. & Bull, L. (2001) A Self-Adaptive Classifier System. In P-L. Lanzi, W. Stolzmann & S.W. Wilson (Eds) *Advances in Learning Classifier Systems: Proceedings of the Third International Workshop*. Springer, pp70-79.
- Hurst, J. & Bull, L. (2006) A Neural Learning Classifier System with Self-Adaptive Constructivism for Mobile Robot Control. *Artificial Life* 12(3): 353 – 380.
- Ingerson, T & Buvel, R. (1984) Structure in Asynchronous Cellular Automata. *Physica D* 10 (1-2): 59-68.
- Kauffman, S.A. (1969) Metabolic Stability and Epigenesis in Randomly Constructed Genetic Nets. *Journal of Theoretical Biology* 22:437-467.
- Kauffman, S. A. (1993) *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford.

Koza, J.R. (1989) Hierarchical Genetic Algorithms Operating on Populations of Computer Programs. In *Proceedings of the 11th Joint International Conference on Artificial Intelligence*. Morgan Kaufmann, pp768-774.

Koza, J.R. (1992) *Genetic Programming*. MIT Press.

Lanzi, P-L & Perrucci, A. (1999) Extending the Representation of Classifier Conditions Part II: From Messy Coding to S-Expressions. In W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela & R.E. Smith (Eds) *Proceedings of the Genetic and Evolutionary Computation Conference – GECCO-99*. Morgan Kaufmann, pp345-352.

Lemke, N., Mombach, J. & Bodmann, B. (2001) A Numerical Investigation of Adaptation in Populations of Random Boolean Networks. *Physica A* 301: 589–600.

Luke, S. & Spector, L. (1996) Evolving Graphs and Networks with Edge Encoding: Preliminary Report. In J.R. Koza (Ed.) *Late Breaking Papers at the Genetic Programming 1996 Conference*, Stanford University, pp117-124.

McCulloch, W.S. & Pitts, W. (1943) A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics* 5: 115-133.

Mesot, B. & Teuscher, C. (2005) Deducing Local Rules for Solving Global Tasks with Random Boolean Networks. *Physica D* 211(1-2):88-106.

Miller, J. (1999) An Empirical Study of the Efficiency of Learning Boolean Functions using a Cartesian Genetic Programming Approach. In W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela & R.E. Smith (Eds) *Proceedings of the Genetic and Evolutionary Computation Conference – GECCO-99*. Morgan Kaufmann, pp1135-1142.

Mitchell, M., Hraber P., & Crutchfield J. (1993) Revisiting the Edge of Chaos: Evolving Cellular Automata to Perform Computations. *Complex Systems* 7: 83-130.

- Nakamura, K. (1974) Asynchronous Cellular Automata and their Computational Ability. *Systems, Computers, Controls* 5(5): 58-66.
- Niehaus, J & Banzhaf, W. (2001) Adaption of operator Probabilities in Genetic Programming. In J. Miller, M. Tomassini, P-L. Lanzi, C. Ryan, A. Tettamanzi & W. Langdon (Eds) *Proceedings of the Fourth European Workshop on Genetic Programming*. Springer, pp325-336.
- Packard, N. (1988) Adaptation Toward the Edge of Chaos. In J. Kelso, A. Mandell & M. Shlesinger (Eds) *Dynamic Patterns in Complex Systems*. World Scientific, pp293-301.
- Poli, R. (1997) Evolution of Graph-like Programs with Parallel Distributed Programming. In T. Back (Ed) *Proceedings of the Seventh International Conference on Genetic Algorithms*. Morgan Kaufmann, pp346-353.
- Poli, R. (1999) Parallel Distributed Genetic Programming, In D. Corne, M. Dorigo & F. Glover (Eds), *New Ideas in Optimisation*. McGraw-Hill, pp403-431.
- Port, R., & van Gelder, T. (1995) *Mind as Motion: Explorations in the dynamics of cognition*. MIT press.
- Pujol, J. & Poli, R. (1998) Efficient Evolution of Asymmetric Recurrent Neural Networks using a PDGP-inspired two-dimensional Representation. In W. Banzhaf, R. Poli, M. Schoenauer & T.C. Fogarty (Eds) *Proceedings of the First European Workshop on Genetic Programming*. Springer, pp130-141.
- Quick, T., Nehaniv, C., Dautenhahn, K. & Roberts, G. (2003) Evolving Embedded Genetic Regulatory Network-Driven Control Systems. In *Proceedings of the Seventh European Artificial Life Conference*. Springer, pp266-277.
- Ray, T. (1992) an Approach to the Synthesis of Life. In C.G. Langton, C. Taylor, J.D. Farmer & S. Rasmussen (Eds) *Artificial Life II*. Addison Wesley, pp371-408.
- Reed, J., Toombs, R. & Barricelli, N. (1967) Simulation of Biological Evolution and Machine Learning: I. Selection of Reproducing Numeric Patterns by Data Processing Machines, Effects of Hereditary Control, Mutation Type and Crossing. *Journal of Theoretical Biology* 17: 319-342.

Sapin, E., Bailleux, O. & Chabrier, J. (2003) Research of a Cellular Automaton Simulating Logic Gates by Evolutionary Algorithms. In *Proceedings of the EvoWorkshops: EuroGP*. Springer, pp414-423.

Schmidt, M. & Lipson, H. (2007) Comparison of Tree and Graph Encodings as Function of Problem Complexity. In *Proceedings of the Genetic and Evolutionary Computation Conference – GECCO-07*. ACM Press, pp1674-1679.

Schwefel, H-P. (1981) *Numerical Optimization of Computer Models*. Wiley.

Sipper, M. (1997) *Evolution of Parallel Cellular Machines*. Springer.

Sipper, M., Tomassini, M. & Capcarrere, S. (1997) Evolving Asynchronous and Scalable Non-uniform Cellular Automata. In *Proceedings of the Third International Conference on Artificial Neural Networks and Genetic Algorithms*. Springer, pp66-70.

Teller, A. & Veloso, M. (1996) Neural Programming and an Internal Reinforcement Policy. In J.R. Koza (Ed.) *Late Breaking Papers at the Genetic Programming 1996 Conference*, Stanford University, pp186-192.

Tomassini, M. & Venzi, M. (2002) Evolution of Asynchronous Cellular Automata for the Density Task. In J. Merelo, P. Adamidis, H-G. Beyer, J-L. Fernandez-Villacanas & H-P. Schwefel (Eds) *Parallel Problem Solving from Nature - PPSN VII*. Springer Verlag, pp934-943.

Teuscher, C. (2002) *Turing's Connectionism*. Springer.

Toth R., Stone, C., De Lacy Costello, B., Adamatzky, A. & Bull, L. (2008) Dynamic Control and Information Processing in the Belousov-Zhabotinsky Reaction using a Co-evolutionary Algorithm. *Journal of Chemical Physics*.

Turing, A. (1948) Intelligent Machinery. See pages 395-432 in [Copeland, 2004].

Turing, A. (1968) Intelligent Machinery. In C.R. Evans & A. Robertson (Eds) *Key Papers: Cybernetics*. Butterworths.

- Valenzuela-Rendon, M. (1991) The Fuzzy Classifier System: a Classifier System for Continuously Varying Variables. In L. Booker & R. Belew (Eds) *Proceedings of the Fourth International Conference on Genetic Algorithms*. Morgan Kaufmann, pp346-353.
- Van den Broeck, C. & Kawai, R. (1990) Learning in Feedforward Boolean Networks. *Physical Review A* 42: 6210-6218.
- Von Neumann, J. (1966) *The Theory of Self-Reproducing Automata*. University of Illinois.
- Wagner, G. & Altenberg, L. (1996) Complex Adaptations and the Evolution of Evolvability. *Evolution* 50:967-97
- Werner, T. & Akella, V. (1997) Asynchronous Processor Survey. *Computer* 30(11): 67-76.
- Wilson, S.W. (1995) Classifier Fitness Based on Accuracy. *Evolutionary Computing* 3: 149-175.
- Wilson, S.W. (2000) Get Real! XCS with Continuous-Valued Inputs. In P-L. Lanzi, W. Stolzmann & S.W. Wilson (Eds) *Learning Classifier Systems: From Foundations to Applications*. Springer, pp209-222.
- Wilson, S.W. (2001) Mining Oblique Data with XCS. In Lanzi, P. L., Stolzmann, W., & Wilson, S. W. (Eds) *Advances in Learning Classifier Systems - IWLCS-2000*. Springer, pp158-176.