Recurrent Cartesian Genetic Programming

Andrew James Turner[1] and Julian Francis Miller[2]

[1] The University of York, England
andrew.turner@york.ac.uk
[2] The University of York, England
julian.miller@york.ac.uk

Abstract. This paper formally introduces Recurrent Cartesian Genetic Programming (RCGP), an extension to Cartesian Genetic Programming (CGP) which allows recurrent connections. The presence of recurrent connections enables RCGP to be successfully applied to partially observable tasks. It is found that RCGP significantly outperforms CGP on two partially observable tasks: artificial ant and sunspot prediction. The paper also introduces a new parameter, *recurrent connection probability*, which biases the number of recurrent connections created via mutation. Suitable choices of this parameter significantly improve the effectiveness of RCGP.

1 Introduction

Cartesian Genetic Programming (CGP) [1] is a form of Genetic Programming (GP) [2] which encodes graph-based computational structures. CGP typically evolves acyclic programs which are only suited to fully observable tasks; when the desired outputs are purely a function of the current inputs. However, many tasks are partially observable and require that previous, as well as current, inputs be considered when calculating outputs. To be applicable to partially observable tasks CGP requires the ability to create programs which hold internal state information; that is to say, some form of memory/feedback. Previously traditional GP has been implemented with memory and feedback using explicit indexed memory [3] and Jordan type architectures [4] respectively.

This paper formally introduces Recurrent Cartesian Genetic Programming (RCGP), an extension to CGP which allows the creation of recurrent / cyclic graphs. RCGP has the ability, through feedback, to store internal state information making it suited to partially observable tasks. Recurrent connections are controlled by a new parameter, *recurrent connection probability*, which defines the likelihood of mutations creating a recurrent connection.

The aim of the paper is to apply and compare CGP and RCGP on partially observable tasks. The study has been undertaken to highlight that there are types of problems for which CGP is currently unsuitable, but to which RCGP can be successfully applied. The aim is not to compare RCGP's performance with other methods suited to partial observable tasks. This is left for further research.

T. Bartz-Beielstein et al. (Eds.): PPSN XIII 2014, LNCS 8672, pp. 476–486, 2014.
© Springer International Publishing Switzerland 2014

The remainder of the paper is organized as follows: Section 2 describes CGP, Section 3 introduces RCGP, Section 4 describes the experiments used to compare CGP and RCGP, Section 5 describes the benchmarks used in the experiments, Section 6 presents the results, Section 7 gives a discussion of the findings and finally Section 8 gives closing conclusions.

2 Cartesian Genetic Programming

CGP [1] [5] is a form of GP [2] which typically evolves acyclic computational structures of nodes (graphs) indexed by their Cartesian coordinates. CGP does not suffer from bloat [6] [7]; a drawback of many GP methods [8]. CGP chromosomes contain non-functioning genes enabling neutral genetic drift during evolution [9] [10]. CGP typically uses point or probabilistic mutation, no crossover and a $(1 + \lambda)$-ES. Although CGP chromosomes are of static size, the number of active nodes varies during evolution enabling variable length phenotypes. The user therefore specifies a *maximum* number nodes, of which only a proportion will be active. Overestimating the number of nodes has shown to greatly aid evolution [11]; which is thought to heighten neutral genetic drift but could also be compensating for length bias [12].

Each CGP chromosome comprises of function genes (F_i), connection genes (C_i) and output genes (O_i). The function genes represent indexes in a function look-up-table and describe the functionality of each node. The connection genes describe from where each node gathers its inputs. For regular acyclic CGP, connection genes may connect a given node to any previous node in the program, or any of the program inputs. The output genes address any program input or internal node and define which are used as program outputs.

Originally CGP programs were organized with nodes arranged in rows (nodes per layer) and columns (layers); with each node indexed by its row and a column. However, this is an unnecessary constraint, as any configuration possible using a given number of rows and columns is also possible using one row with many columns; provided the total number of nodes remains constant. This is due to CGP being capable of evolving where each node connects its inputs. Consequently, here the chromosomes are defined with one row and n columns; with each node only indexed by its column. A generic (one row) CGP chromosome is given in Equation 1; where α is the arity of each node, n is the number of nodes and m is the number of program outputs.

$$F_0 C_{0,0} ... C_{0,\alpha} F_1 C_{1,0} ... C_{1,\alpha} \text{ ...... } F_n C_{n,0} ... C_{n,\alpha} O_0 ... O_m \quad (1)$$

An example CGP program is given in Figure 1 along with its corresponding chromosome. As can be seen, all nodes are connected to previous nodes or program inputs. Not all program inputs have to be used, enabling evolution to decide which inputs are significant. An advantage of CGP over tree-based GP, again seen in Figure 1, is that node outputs can be reused multiple times, rather than requiring the same value to be recalculated if it is needed again. Finally, not all nodes contribute to the final program output, these represent the inactive nodes which enable neutral genetic drift and make variable length phenotypes possible.

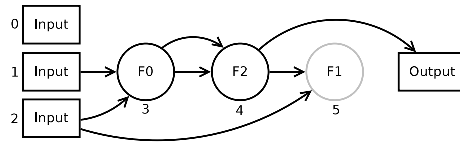


Fig. 1. Example CGP program corresponding to the chromosome: 012 233 124 4

3 Recurrent Cartesian Genetic Programming

Although RCGP has never been formally presented, it has been previously discussed as a possible extension to CGP [1]. RCGP has also been used as a method for removing length bias[1] [12] when investigating why CGP does not suffer from bloat [7]. Additionally a form of CGP has been used which implemented a Jordan type architecture [13] for allowing feedback [14]. Here the application was Cartesian Genetic Programming of Artificial Neural Networks (CGPANN) [15,16]; a NeuroEvolutionary technique based on CGP. Although using Jordan type architectures represents a simple method for allowing recurrent connections, it does so in a very restricted form. For instance, the user must decide in advance how many and what type of recurrent connections will be used.

3.1 Recurrent Cartesian Genetic Programming Implementation

In CGP, connection gene values are restricted so as to only allow acyclic connections. In RCGP this restriction is lifted so as to allow connections between a given node and *any other* node in the program (including itself) or program inputs. An example program which could be generated using RCGP is given in Figure 2 along with the corresponding chromosome.

RCGP phenotypes are executed similar to CGP phenotypes. Starting at the active node closest to the inputs, each node calculates its output value based on its inputs. Once all active nodes have been updated, the program outputs are recorded. However, with the presence of recurrent connections, a nodes output can be required before it has been calculated. To deal with this, all the nodes are initialised to output zero until they calculate their own value. Therefore when executing a RCGP phenotype the the following process is used:

1. set all active nodes to output zero
2. apply the next set of program inputs
3. update all active nodes **once** from program inputs to program outputs
4. read the program outputs
5. repeat from 2 until all program input sets have been applied

It should be noted that the program outputs are read *once* for each set of applied program inputs. It would also be possible to execute the program multiple

[1] Although through email correspondence with Brian Goldman it may be the case that RCGP only serves to alter the length bias rather than remove it.

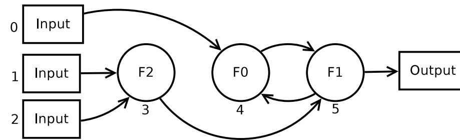


Fig. 2. Example RCGP program cosponsoring to the chromosome: 212 005 134 5

times for each set of program inputs. In such a case, the average of the program outputs or the settled program outputs[2] could be taken. The method described here was chosen for its simplicity and because there is no guarantee that the program outputs would ever settle.

A drawback of placing no constraints on connection gene values is that, on average, mutations to connection genes will result in as many feed-forward connections as recurrent. As it is highly unlikely that many tasks will require fifty percent of connections to be recurrent, this places a bias towards possibly unsuitable areas in the solution space. For this reason, a new parameter is introduced which controls the likelihood of mutations creating recurrent connections. This parameter is called *recurrent connection probability*. A recurrent connection probability of zero percent results in only feed-forward connections (i.e. regular CGP). A recurrent connection probability of fifty percent results in mutations causing as many feed-forward connections as recurrent (i.e. RCGP without the new parameter). A recurrent connection probability of one hundred percent results in only recurrent connections. It should be noted that this parameter does not directly control the number of recurrent connections, only the probability of mutations creating recurrent connections.

An important property of CGP is that the active nodes can be determined before executing the program. This is significant as a high proportion of nodes are often inactive [11] and calculating their outputs wastes computation time. To determine which nodes are active the following algorithm is used [1]: 1) add each program output node to a list of active nodes 2) for each node added to the active node list, add the nodes to which they also connect 3) continue until the program inputs are reached. Determining the active nodes for RCGP follows a similar algorithm except only nodes which are not currently in the active node list are added. This extra criteria breaks cycles enabling active nodes to be determined for RCGP.

3.2 Implications of Recurrent Connections

An implication of RCGP is that it is now possible for chromosomes to describe phenotypes where none of the active nodes connect to the program inputs. These programs are therefore unsuited to any realistic task. However such programs will likely score a low fitness and be quickly dropped from the population.

[2] Where settled output refers to the converged program output value(s) after many updates of the active nodes whilst applying the same program inputs.

Another implication of allowing recurrent connections occurs when applying RCGP to tasks where each set of inputs are independent from each other. For example, suppose we are trying to evolve a program that can implement a six bit parity circuit. Normally, we think of each line of the truth table as being independent of one another (i.e. the order in which each set of inputs occurs is unimportant). If the fitness function always tests each line of the truth table in the same order, RCGP could in principle use previous inputs to "predict" the correct output. This was shown to be the case in [7]. In an extreme case, RCGP could "predict" the correct outputs without ever considering the program inputs[3]. It is therefore important that RCGP should only be applied to tasks where the series of inputs are related, such as in time series prediction; otherwise additional precautions would be required to prevent this behaviour.

4 Experiments

The experiments presented are designed to test if RCGP is a suitable extension to CGP when solving partially observable tasks. As RCGP is implemented using the recurrent connection probability, this parameter is varied over [0, 10, 20, 50] percent. Where zero percent is equivalent to CGP, fifty percent is equivalent to RCGP without the additional parameter and ten and twenty percent represents RCGP with lower biases for recurrent connections.

If RCGP achieves statistically significantly better fitness than CGP on the given tasks, then RCGP will be considered a suitable extension to CGP when solving partially observable problems. If biasing the level of recurrence is shown to statistically significantly influence fitness, then the recurrent connection probability will be considered a suitable parameter for RCGP.

As this is the first time RCGP has been investigated it is unknown how the number of available nodes will influence results. For this reason each experiment is repeated over a range of available nodes [10, 20, ..., 90, 100] to ensure a fair comparison between CGP and RCGP. Other than the parameters previously given, the following are used throughout the experiments: $(1+4)$-ES, 3% probabilistic mutation and a node arity of two. The results presented are the average fitness of fifty independent runs. Each run is given ten thousand generations before terminating the search.

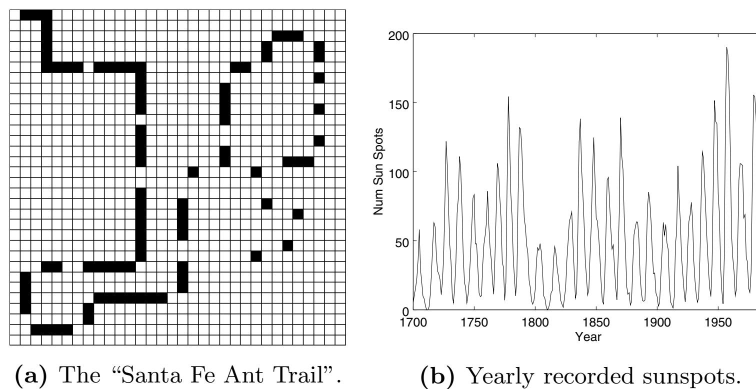
5 Benchmarks

Two partially observable benchmarks are used in the described experiments, Artificial Ant and Sunspots. The Artificial Ant benchmark is a reinforcement learning control task and the Sunspots benchmark is a supervised learning series forecasting task.

[3] This was shown to be the case in unpresented results where RCGP "solved" the six bit parity task without any inputs!

5.1 Artificial Ant

The Artificial Ant problem [17] is a classic challenging [18] benchmark commonly used by GP [2]. The task is to design a controller which navigates an ant around a toroidal map maximising food intake. The ant can only perceive whether the location ahead of is current position contains food. Each time step the ant undertakes one of four actions: move forward, turn left 90°, turn right 90° or do nothing. The map used here is the "Santa Fe Ant Trail" [2] given in Figure 3.



(a) The "Santa Fe Ant Trail". **(b)** Yearly recorded sunspots.

Fig. 3. (a) Depiction of the "Santa Fe Ant Trail". Black and white represent food and no food respectively. (b) Yearly number of recorded sunspots.

In this paper the form of the controller differs from that commonly used by GP [2]. Here the evolved program's inputs describe if the location ahead contains food and the program's outputs are decoded into one of the possible four actions; this is not dissimilar to the original implementation [17]. Other GP implementations [2] create programs where the program inputs are the possible actions and the program outputs are unused. The function set used by the nodes causes the inputs (actions) to either be implemented outright or to be conditional on whether food is ahead. Once the program outputs are reached the program starts over. CGP has previously been applied to the benchmark in its more commonly used form [5].

In this paper, the evolved controllers have two mutually exclusive inputs, whereby the first input is set as '1' if the location ahead of the ant contains food, else it is set as '0'. The controller has two outputs, where: [1 1] represents move forward, [0 1] turn right, [1 0] turn left and [0 0] do nothing. The ant starts in the top left $(0, 0)$ of the toroidal map facing east and is allowed 400 time steps to consume as much food as possible. The amount of food eaten is then used as the fitness measure; out of a maximum 89. The function set used comprises of the four Boolean logic gates: AND, OR, NOT, and XOR.

5.2 Sunspots

The Sunspots benchmark [19] is a commonly used [20] time series prediction benchmark which describes the number of observed sunspots dating back to 1700. The data was recorded by the SIDC-team, at the World Data Center for the Sunspot Index, Royal Observatory of Belgium [19]. The dataset contains the yearly number of recorded sunspots between 1700 and 1987; given in Figure 3. The first 221 years (1700-1920) are used as the training set with the remaining 67 years (1921-1987) used as the testing set.

Most series forecasters which are applied to the Sunspots benchmark use multiple inputs consisting of the current and previous years number of sunspots. However, in this paper only one input is used which gives the current number of sunspots. This restriction to one input was imposed to force the task to become partially observable. This restriction also makes the task much more challenging since any trends in the data must be calculated internally as the data is passed in year by year. The single output is the predicted number of sunspots 35 years ahead of the current input. The single input to the series forecaster is normalised into a $[0, 1]$ range by dividing by two hundred (a value greater than the highest number of sunspots in any observed year). The single output is also multiplied by two hundred before being used as the predicted number of sunspots.

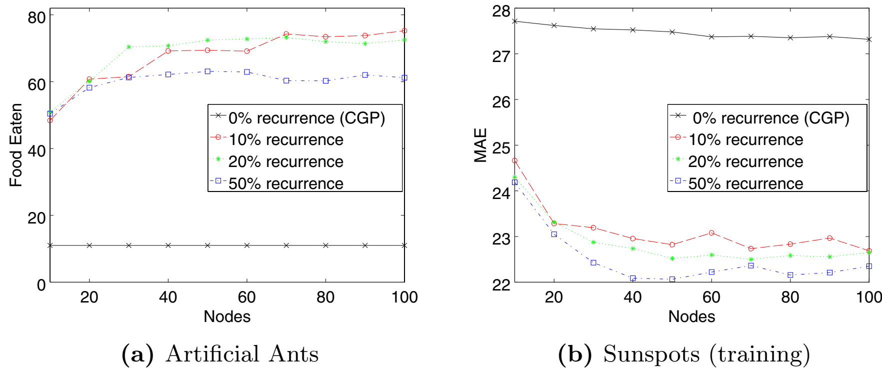
The fitness measure is the mean average error (MAE) given by: $\frac{1}{N}\sum_{i=1}^{N}|e_i|$ where N is the number of samples and e is the difference between the actual and predicted number of sunspots. The function set used for this task comprises of ten symbolic expressions: $x_1 + x_2$, $x_1 - x_2$, $x_1 \times x_2$, $x_i \div x_j$, $|x_1|$, x_1^2, x_1^3, e^{x_1}, $\sin(x_1)$ and $\cos(x_1)$. Where x_1 and x_2 are the two inputs to each node and the division operator is protected so as to return one when dividing by zero.

6 Results

The the average fitnesses (from 50 runs) achieved using RCGP are given for the Artificial Ant and Sunspots benchmarks in Figure 4. It should be recalled that a recurrent connection probability of zero percent is equivalent to regular CGP. The average generalisation performance on the Sunspot testing set is also given in Figure 5 along with an example forecaster created using one hundred nodes and a recurrent connection probability of ten percent.

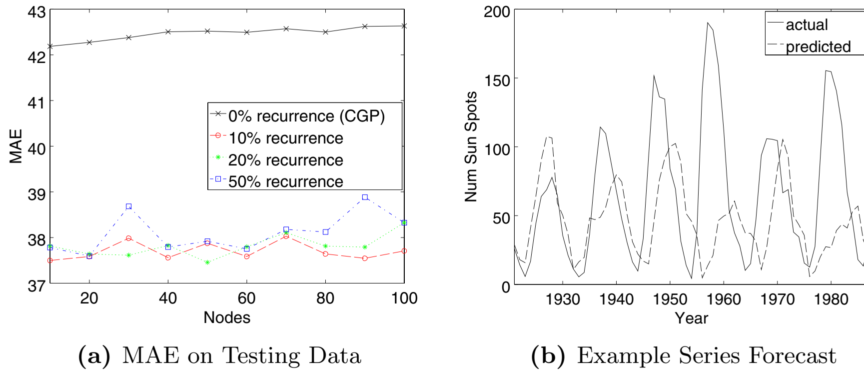
To identify if the differences due to the recurrent connection probability seen in Figure 4 are statistically significant the results are analysed using the non-parametric two sided Mann-Whitney U-test. When using the U-test $p<0.05$ indicates statistical significance between two sets of data. Tables 1 and 2 give the p values when comparing pairs of recurrent connection probabilities using ten, twenty and one hundred nodes for the Artificial Ant and Sunspot (training set) benchmarks.

As can be seen in Figure 4 and Tables 1 and 2, a recurrent connection probability of zero percent consistently performs worst on both benchmarks with statistical significance. This demonstrates that there are types of tasks which regular CGP is not suited to but to which RCGP can be successfully applied.



(a) Artificial Ants **(b)** Sunspots (training)

Fig. 4. Results of applying RCGP on the two benchmark



(a) MAE on Testing Data **(b)** Example Series Forecast

Fig. 5. Generalisation of RCGP on the Sunspot test set

When comparing recurrent connection probabilities, it can be seen that lower percentages (ten and twenty) give statistically significantly better results on the Artificial Ant benchmark than higher percentages (fifty). On the Sunspots benchmark the opposite is true, with higher levels producing the best results on the training set. However, this greater performance on the training set is accompanied by weaker generalisation on the testing set; seen in Figure 5.

7 Discussion

The results given in Section 6 clearly demonstrate that CGP is unsuitable for partially observable tasks. This is not a surprising result as CGP has no capacity to recall previous inputs or infer internal state information. For instance the best strategy CGP could find for the Artificial Ant task was to rotate until food is ahead, and then move forward.

Table 1. Artificial Ant: p values comparing pairs of recurrent connection probabilities

	0%	10%	20%	50%
0%	1	**∼0**	**∼0**	**∼0**
10%	-	1	0.274	0.118
20%	-	-	1	0.576
50%	-	-	-	1

(a) 10 Nodes

	0%	10%	20%	50%
0%	1	**∼0**	**∼0**	**∼0**
10%	-	1	0.391	**0.025**
20%	-	-	1	**0.002**
50%	-	-	-	1

(b) 50 Nodes

	0%	10%	20%	50%
0%	1	**∼0**	**∼0**	**∼0**
10%	-	1	0.260	**2E-5**
20%	-	-	1	**2E-4**
50%	-	-	-	1

(c) 100 Nodes

Table 2. Sunspots: p values comparing pairs of recurrent connection probabilities

	0%	10%	20%	50%
0%	1	**∼0**	**∼0**	**∼0**
10%	-	1	**0.022**	**0.021**
20%	-	-	1	0.759
50%	-	-	-	1

(a) 10 Nodes

	0%	10%	20%	50%
0%	1	**∼0**	**∼0**	**∼0**
10%	-	1	0.234	**0.011**
20%	-	-	1	0.167
50%	-	-	-	1

(b) 50 Nodes

	0%	10%	20%	50%
0%	1	**∼0**	**∼0**	**∼0**
10%	-	1	0.671	0.312
20%	-	-	1	0.458
50%	-	-	-	1

(c) 100 Nodes

The results in Section 6 show that RCGP *is* highly suited for partially observable tasks. For both benchmarks it dramatically and statistically significantly outperforms CGP.

Section 6 also showed that simply allowing mutation to create feed-forward or recurrent connections with equal probability does not always produce the best results. This is because it is unlikely that a given task will require fifty percent of connections to be recurrent. The introduction of the recurrent connection probability parameter allows the user to bias mutations so as to produce greater or fewer recurrent connections. It has been shown that using a recurrent connection probability of fifty percent (i.e. effectively not using the recurrent connection probability parameter) produces poor results on the Artificial Ant benchmark and causes over training on the Sunspot benchmark[4]. Using a recurrent connection probability of ten percent produced the best results on the Artificial Ant benchmark and produce the best MAE on the testing set for the Sunspot benchmark. The recurrent connection probability is therefore an important additional parameter when using RCGP.

8 Conclusion

RCGP is an extension to CGP which enables application to partially observable tasks. On two partially observable benchmark problems, Artificial Ant and Sunspot prediction, RCGP gives statistically significant improvements compared with acyclic CGP. RCGP has been implemented using a recurrent connection probability parameter which biases the number of recurrent connections created

[4] Over training could be controlled via the use of a validation set but this was not considered here.

via mutations. This is introduced as simply allowing mutations to connect any two nodes creates an unhelpful bias for as many feed-forward connections as recurrent. Further research is needed to compare the performance of RCGP with other methods suited to partially observable problems and to apply RCGP to additional domains; such as creating recurrent artificial neural networks.

References

1. Miller, J.F. (ed.): Cartesian Genetic Programming. Springer (2011)
2. Koza, J.R.: Genetic Programming: On the programming of computers by means of natural selection, vol. 1. MIT Press (1992)
3. Teller, A.: Turing completeness in the language of genetic programming with indexed memory. In: IEEE Evolutionary Computation, pp. 136–141 (1994)
4. Teredesai, A., Govindaraju, V., Ratzlaff, E., Subrahmonia, J.: Recurrent genetic programming. In: 2002 IEEE International Conference on Systems, Man and Cybernetics, vol. 4, pp. 5–9. IEEE (2002)
5. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: Poli, R., Banzhaf, W., Langdon, W.B., Miller, J., Nordin, P., Fogarty, T.C. (eds.) EuroGP 2000. LNCS, vol. 1802, pp. 121–132. Springer, Heidelberg (2000)
6. Miller, J.F.: What bloat? Cartesian genetic programming on Boolean problems. In: Genetic and Evolutionary Computation Conference, pp. 295–302 (2001)
7. Turner, A.J., Miller, J.F.: Cartesian Genetic Programming: Why No Bloat? In: Genetic Programming: 17th European Conference (to appear, 2014)
8. Silva, S., Costa, E.: Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories. Genetic Programming and Evolvable Machines 10(2), 141–179 (2009)
9. Vassilev, V.K., Miller, J.F.: The Advantages of Landscape Neutrality in Digital Circuit Evolution. In: Miller, J.F., Thompson, A., Thompson, P., Fogarty, T.C. (eds.) ICES 2000. LNCS, vol. 1801, pp. 252–263. Springer, Heidelberg (2000)
10. Yu, T., Miller, J.F.: Neutrality and the evolvability of boolean function landscape. In: Miller, J., Tomassini, M., Lanzi, P.L., Ryan, C., Tetamanzi, A.G.B., Langdon, W.B. (eds.) EuroGP 2001. LNCS, vol. 2038, pp. 204–217. Springer, Heidelberg (2001)
11. Miller, J.F., Smith, S.: Redundancy and computational efficiency in Cartesian genetic programming. Evolutionary Computation 10(2), 167–174 (2006)
12. Goldman, B.W., Punch, W.F.: Length bias and search limitations in Cartesian genetic programming. In: Proceeding of the Fifteenth Annual Conference on Genetic and Evolutionary Computation Conference, pp. 933–940. ACM (2013)
13. Jordan, M.I.: Serial order: A parallel distributed processing approach. Technical report, Institute for Cognitive Science (1986)
14. Khan, M., Khan, G., Miller, J.: Efficient representation of recurrent neural networks for markovian/non-markovian non-linear control problems. In: IEEE Intelligent Systems Design and Applications, pp. 615–620 (2010)
15. Khan, M.M., Ahmad, M.A., Khan, M.G., Miller, J.F.: Fast learning neural networks using Cartesian Genetic Programming. Neurocomputing 121, 274–289 (2013)
16. Turner, A.J., Miller, J.F.: Cartesian Genetic Programming encoded Artificial Neural Networks: A Comparison using Three Benchmarks. In: Genetic and Evolutionary Computation, pp. 1005–1012 (2013)

17. Jefferson, D., Collins, R., Cooper, C., Dyer, M., Flowers, M., Korf, R., Taylor, C., Wang, A.: The genesys system: Evolution as a theme in artificial life. In: Artificial Life. Addison-Wesley, Redwood City (1990)
18. Langdon, W.B., Poli, R.: Why ants are hard. Technical report, School of Computer Science, The University of Birmingham, Birmingham, UK (1998)
19. SIDC-team: The International Sunspot Number. Monthly Report on the International Sunspot Number, online catalogue (1700-1987)
20. Khashei, M., Bijari, M.: An artificial neural network (p,d,q) model for timeseries forecasting. Expert Systems with Applications 37(1), 479–489 (2010)