Recurrent Cartesian Genetic Programming Applied to Famous Mathematical Sequences

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

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

Abstract. Recurrent Cartesian Genetic Programming (RCGP) is a recent extension to Cartesian Genetic Programming (CGP) which allows CGP to create recurrent programs. This paper investigates using RCGP to create recurrent symbolic equations which produce famous mathematical sequences. The use of RCGP is contrasted with using standard CGP which can only produce explicit solutions. The results demonstrate that RCGP is capable of producing recursive equations for all the given sequences whereas CGP cannot always produce explicit equations. It is also discussed that since RCGP is a superset of CGP it should also be capable of finding explicit equations. Finally recommendations concerning the initial conditions of RCGP programs are also discussed.

1 Introduction

There are numerous mathematical sequences [12] which have been of interest to mathematicians for many years; as early as Archimedes (287 BC) where mathematical sequences were used in his "Method of Exhaustion" [13]. Mathematical sequences can be defined in two forms, either explicitly or recursively. An explicit equation returns the n^{th} value in a sequence when passed the value of n and a recursive equation returns the n^{th} value in a sequence on the n^{th} iteration.

Cartesian Genetic Programming (CGP) [6] is a form of Genetic Programming (GP) [4] which has previously being used to map explicit symbolic equations to sequences of numbers [6, 8, 18]. Standard CGP has always been used to produce explicit solutions as it is not capable of creating recurrent equations. However, a recent extension to CGP, Recurrent Cartesian Genetic Programming [16], now enables CGP to create arbitrary programs containing recurrence; including recurrent equations. This paper investigate applying CGP and RCGP to a number of famous mathematical sequences in order to describe them using explicit *and* recurrent symbolic equations. A separate extension to CGP, Self Modifying CGP, has also previously been used to produce what could described as recurrent solutions [2]. There the CGP chromosomes were able to reconfigure themselves when executed in order to produced different behaviours upon future executions.

The remainder of the paper is structured as follows: Sections 2 and 3 describe CGP and RCGP respectively, Section 4 describes the experiments presented along with the famous mathematical sequences used, Section 5 gives the results of the experiments and finally Sections 6 and 7 give a discussion and present our conclusions.

2 Cartesian Genetic Programming

CGP [6, 8] is a form of GP [4] which typically evolves acyclic computational structures of nodes (graphs) indexed by their Cartesian coordinates. CGP does not suffer from bloat[3] [5, 15]; a drawback of many GP methods [11]. CGP chromosomes contain non-functioning genes enabling neutral genetic drift during evolution [17]. 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 (solutions). 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 [7]; which is thought to heighten neutral genetic drift.

Each CGP chromosome is comprised 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 the locations from which each node gathers its inputs. For regular acyclic CGP, connection genes may connect a given node to any previous node in the graph, or any of the program inputs. The output genes address any program input or internal node and define which are used as program outputs.

An example of a generic 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. An example CGP program is given with its corresponding chromosome in Figure 1. 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.

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

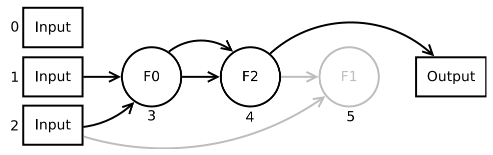


Fig. 1: Example CGP program corresponding to the chromosome: 012 233 124 4

[3] Bloat is a phenomenon seen in many GP methods where the size of the evolved programs grow continuously during evolution with little or no improvement in fitness.

3 Recurrent Cartesian Genetic Programming

In regular CGP chromosome connection genes are restricted to only allow connections to previous nodes in the graph. In RCGP this restriction is lifted so as to allow connections genes to connect a given node to *any other* node (including itself) or program inputs. Once the acyclic restriction is removed, RCGP solutions can contain recurrent connections or feedback. An example RCGP program is given in Figure 2 along with its corresponding chromosome. Another simpler but less flexible method of using CGP to create recurrent programs is to enforce a Jordan type architecture [9]; where program outputs are fed back as inputs. A slightly more complex multi-chromosome version of CGP has also been adapted to be capable of creating transistor circuits which contain cyclic connections [19].

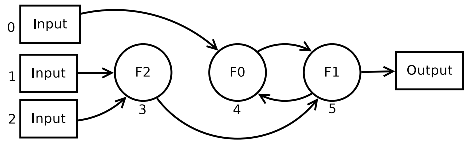


Fig. 2: Example RCGP program cosponsoring to the chromosome: 212 005 134 5

As described in [16], placing no restriction on connections genes results in mutations creating as many recurrent as feed-forward connections. However, it is likely that most problems do not require fifty percent of the connections to be recurrent. For this reason a new parameter was introduced called recurrent connection probability. This parameter controls the probability that a connection gene mutation will create a recurrent connection.

RCGP chromosomes are interpreted identically to standard CGP. The inputs are applied, each node is updated in order of index, and the outputs are read. The next set of inputs are then applied and the process repeated. One important aspect of RCGP, again described in [16], is that it is now possible for a node's output value to be read before it has been calculated. Here, as described in [16], all nodes are set to output zero until they have calculated their own output value. This is akin to the initial conditions in recursive equations.

4 Experiments

The experiments presented investigate if CGP and RCGP can be used to create explicit and recurrent equations which describe a given sequence. The parameters used by CGP and RCGP are given in Table 1. Although it is known that that using a high number of nodes aids the evolutionary search [7], here a low number of nodes are used so the evolved equations can be easily inspected i.e. are small.

All experiments are implemented and run using the development version[4] of the cross platform open source CGP-Library [14]. A custom fitness function

[4] The development version allows for recurrent connections whereas the current stable release does not.

Table 1: CGP/RCGP Parameters

Parameter	Value
Evolutionary Strategy	(1+4)-ES
Max Generations	1,000,000
Nodes	20
Node Arity	2
Mutation Type	probabilistic
Mutation Rate	5%
Recurrent Connection Probability	10% (for RCGP)
Function Set	add, sub, mul, div

is used with the CGP-Library when assessing chromosome fitness. This fitness function awards a score equal to the number of values in the training sequence (100), minus the number of correct values predicted in sequence, minus 0.01 for any further correct values after an incorrect value was predicted. Therefore lower fitness represent a fitter solution.

When using CGP, the input to the chromosomes is n and the expected output is the n^{th} value in the sequence. When using RCGP the input is fixed at the value of one and the chromosome is updated multiple times to produce a sequence of numbers. When the chromosome is updated n times it should produce, in order, the first n values in the sequence. It would also be possible to use RCGP and input the value n instead of the constant one. In this case RCGP could produce explicit as well as recurrent solutions. Here however, RCGP was forced to produce recurrent solutions.

The famous mathematical sequences used by the experiments are now introduced.

4.1 Hexagonal Numbers

The Hexagonal number sequence, A000384 from [12], is the number of dots which make up a sequence of hexagons and all the hexagons it contains; see Figure 3. It is defined explicitly by Equation 2 where $n \geq 1$. This produces the following sequence: 1,6,15,28,45,66,91,120,153,190,...

$$y(n) = \frac{2n(2n-1)}{2} \tag{2}$$

4.2 Lazy Caterers Sequence

The Lazy Caterers Sequence (or more formally the central polygonal numbers), A000124 from [12], is the number of pieces a cake can be divided into with n cuts. The sequence is shown graphically in Figure 3 and described explicitly by Equations 3; where $n \geq 0$. This produces the following sequence: 1,2,4,7,11,16,22,29,37,46,...

$$y(n) = \frac{n^2 + n + 2}{2} \tag{3}$$

4.3 Magic Constants

The sequence of Magic Constants, A006003 from [12], are the values each row, column and diagonal of a $n \times n$ magic square[5] can sum to. The magic squares corresponding to n=3, 4 and 5 are given in Figure 3. The sequence is described explicitly by Equations 4; where $n \geq 1$. This produces the following sequence: 1,5,15,34,65,111,175,260,369,505,...

$$y(n) = \frac{n(n^2 + 1)}{2} \tag{4}$$

4.4 Fibonacci

The Fibonacci sequence, A000045 from [12], is such that each value is the sum of the previous two value; where the first two values are set as one. The sequence is described explicitly in Equation 5, but is more commonly given recursively such as in Equation 6. This produces the following sequence: 1,1,2,3,5,8,13,21,34,55,...

$$y(n) = \frac{(1 + \sqrt{5})^n - (1 - \sqrt{5})^n}{2^n \sqrt{5}} \tag{5}$$

$$y(n) = \begin{cases} 1, & \text{if } n \leq 1 \\ y(n-1) + y(n-2), & \text{otherwise} \end{cases} \tag{6}$$

5 Results

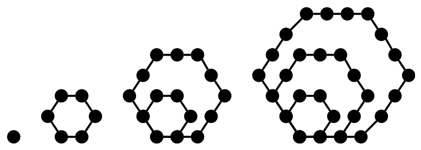
The results of applying CGP and RCGP towards producing number sequences are now presented. All of the results given are the average of fifty independent runs.

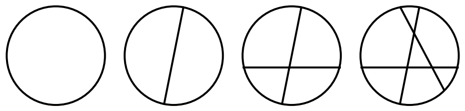
5.1 Hexagonal

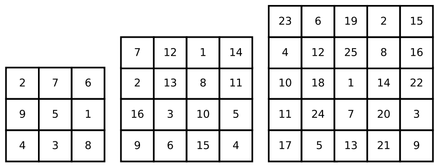
CGP found an explicit solution to the Hexagonal sequence in all of the fifty runs using an average of 2,481 evaluations[6]. The solutions found used an average of 5.44 active nodes. An example explicit solution found using CGP is given in Figure 4. RCGP found a recursive solution to the Hexagonal sequence in all of the fifty runs using a average of 39,279 evaluations. The solutions found used an average of 10.60 active nodes. An example recurrent solution found using RCGP is given in Figure 4.

[5] A magic square is an $n \times n$ grid of numbers where the sum of each row, column and diagonal is the same value.

[6] The number of evaluations is the number of solutions (chromosomes) evaluated before a solution is found.


(a) Hexagonal Numbers (n=1,2,3,4)


(b) Lazy Caterers Numbers (n=0,1,2,3)



2	7	6
9	5	1
4	3	8

7	12	1	14
2	13	8	11
16	3	10	5
9	6	15	4

23	6	19	2	15
4	12	25	8	16
10	18	1	14	22
11	24	7	20	3
17	5	13	21	9

(c) Magic Constants (n=3,4,5)

Fig. 3: Number sequences given graphically.

5.2 Lazy Caterer

CGP did not find an explicit solution to the Lazy Caterer sequence in any of the fifty runs. However, an explicit solution, using CGP was found in a longer run and is given in Figure 5[7]. RCGP found a recursive solution to the Lazy Caterer sequence in 48 of the 50 run using a average of 7,626 evaluations. The solutions found used an average of 10.53 active nodes. An example recurrent solution found using RCGP is given in Figure 5.

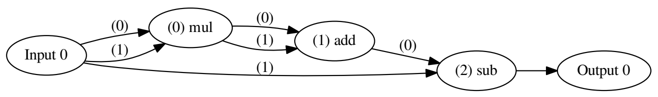
5.3 Magic Constants

CGP found an explicit solution to the Magic Constants sequence in all of the fifty runs using an average of 557,592 evaluations. The solutions found used an average of 8.52 active nodes. An example explicit solution found using CGP is given in Figure 6. RCGP found a recursive solution to the Magic Constants sequence in 43 of the 50 runs using a average of 686,929 evaluations. The solutions found used an average of 12.79 active nodes. An example recurrent solution found using RCGP is given in Figure 6.

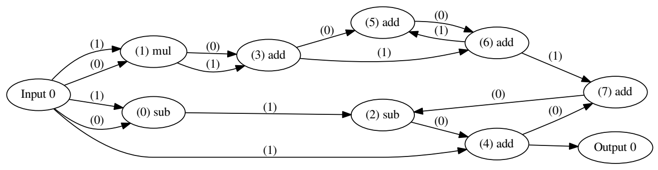
5.4 Fibonacci

CGP could not find an explicit solution to the Fibonacci sequence in any of the fifty runs. RCGP found a recursive solution to the Fibonacci sequence in all of

[7] Found by repeatedly running CGP with 10,000,000 generations until a solution was found.

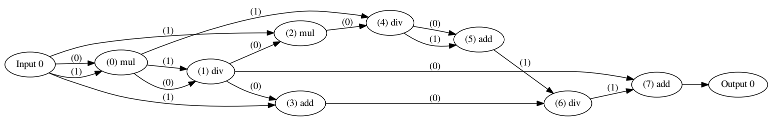


(a) CGP

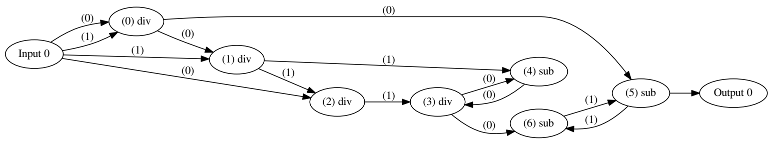


(b) RCGP

Fig. 4: Example CGP and RCGP Hexagonal solutions.



(a) CGP



(b) RCGP

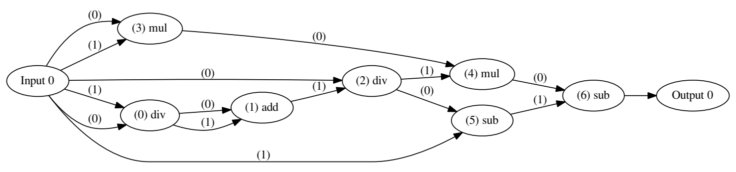
Fig. 5: Example CGP and RCGP Lazy Caterer solutions.

the fifty runs using a average of 27,075 evaluations. The solutions found used an average of 12.00 active nodes. An example recurrent solution found using RCGP is given in Figure 7.

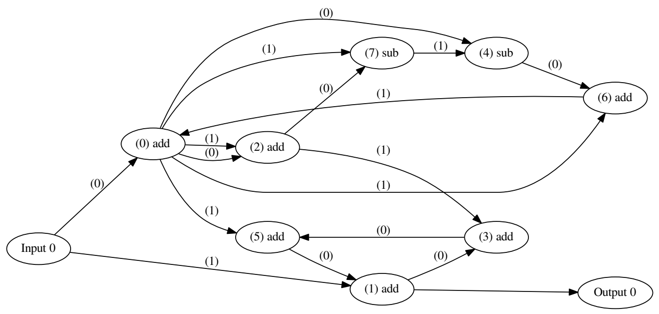
As other GP methods have also been previously applied to the Fibonacci sequence a comparison can be made. However it should be noted that the implementations used are very different between methods e.g. the length of the sequences used, the use of training and testing sets, the percentage of runs which found a solution. Therefore only a very superficial comparison can be made. The results of this comparison are given in Table 2 where RCGP is shown to be very competitive.

6 Discussion

For all of the sequences investigated RCGP managed to find recurrent solutions for the majority of runs. CGP however failed to find explicit solutions to the Lazy Caterers sequence and the Fibonacci sequence. It is unsurprising that CGP could



(a) CGP



(b) RCGP

Fig. 6: Example CGP and RCGP Magic Constants solutions.

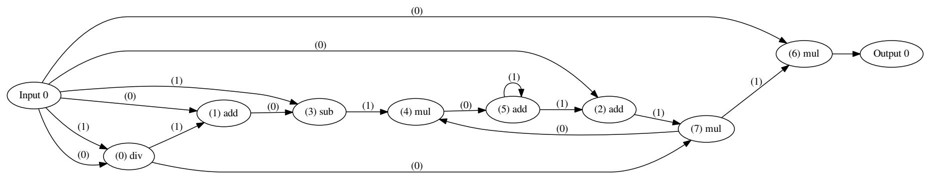


Fig. 7: Example RCGP Fibonacci solution.

not find a explicit solution for the Fibonacci sequence as it cannot be easily defined explicitly. However, it was surprising that CGP failed to find explicit solutions to the Lazy Caterers sequence which can easily be defined explicitly. It is likely that the task was harder than anticipated and CGP required more nodes and more generations to find solutions. It was also shown for the Fibonacci sequence that RCGP produces very competitive results compared to other GP techniques.

Interestingly not all sequences have explicit forms, for instance many chaotic sequences do not. Therefore there will be sequences which can never be described using standard CGP. It is also likely to be true that certain sequences are more easily described explicitly or recursively. For instance CGP found explicit solutions much faster for the Hexagonal sequence, whereas RCGP found recurrent solutions much more easily for the Lazy Caterers sequence; which can both be easily defined explicitly or recursively. A benefit of RCGP, not explored in this paper, is that RCGP can create acyclic *and* cyclic programs. In the research presented here RCGP was forced to produce recurrent solutions, by fixing the input, but it is generally capable of producing feed-forward and recurrent so-

Table 2: GP methods: Fibonacci Sequence

Method	Evaluations
RCGP	27,075
Multi-niche Genetic Programming [10]	200,000
Probabilistic Adaptive Mapping Developmental GP [20]	212,000
SMCGP [2]	1,000,000
Machine Language Programs [3]	1,000,000
Object Oriented GP [1]	20,000,000

lutions. Therefore in situations where it is was not known whether an explicit solution is possible, RCGP could be applied. If a given sequence is more easily described explicitly then this should, in theory, produce an evolutionary pressure to produce explicit solutions. If a given sequence is more easily described recursively then RCGP is also capable of finding a recursive solution.

It was noticed that many solutions found contained addition nodes with their outputs fed back as an input; such as node (5) in Figure 7. As all nodes are initialised to output zero before they calculate their own output value, this has the effect of implementing a summation; where the output of node (5) is the running sum of all the previous outputs of node (4). This interesting behaviour also hold for subtraction. However it does not hold for multiplication as the node is also initialised to output zero; if a multiplication node's output were fed back as an input it would forever output zero. It is therefore possible that simpler recurrent equations could be formed if multiplication nodes could be used to store the product of previous inputs; akin to how addition nodes store the summation. This can be achieved by initialising multiplication nodes to output *one* until they have calculated their own output value. Then multiplication nodes could be arranged such that they produce the product of previous inputs; the same would also be true for division. It is therefore recommended that future developments of RCGP should initialise addition and subtraction nodes to output zero and multiplication and division nodes to output one. It may also be the case that other node functions benefit from being initialised to specific values and this should be considered when extending the function set.

7 Conclusion

This paper has demonstrated the use of RCGP for producing recurrent symbolic equations which describe famous mathematical sequences. It has been shown that CGP is only capable of producing explicit solutions. It has been shown that RCGP is capable of producing recursive solutions and is also capable of producing explicit solutions. It is therefore concluded, given that not all sequences have explicit forms, that RCGP is a more general solution to producing symbolic equations which describe mathematical sequences.

It was also identified that RCGP often arranged addition nodes so as to implement efficient summation operations. Currently, due to all nodes being initialised to output zero, this ability does not extend to multiplication nodes producing product operations. Therefore it is recommended that in future work multiplication nodes should be initialised to output *one* enabling multiplication nodes to produce product operations.

References

1. Agapitos, A., Lucas, S.M.: Learning recursive functions with object oriented genetic programming. In: EuroGP'06. pp. 166–177 (2006)
2. Harding, S., Miller, J.F., Banzhaf, W.: Self Modifying Cartesian Genetic Programming: Fibonacci, Squares, Regression and Summing. In: EuroGP'09. pp. 133–144 (2009)
3. Huelsbergen, L.: Learning recursive sequences via evolution of machine-language programs. In: GP'97. p. 186194 (1997)
4. Koza, J.R.: Genetic Programming: vol. 1, On the programming of computers by means of natural selection, vol. 1. MIT press (1992)
5. Miller, J.F.: What bloat? Cartesian genetic programming on Boolean problems. In: GECCO'01. pp. 295–302 (2001)
6. Miller, J.F.: Cartesian Genetic Programming. Springer (2011)
7. Miller, J.F., Smith, S.: Redundancy and computational efficiency in Cartesian genetic programming. Evolutionary Computation 10(2), 167–174 (2006)
8. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: EuroGP'00. vol. 1820, pp. 121–132 (2000)
9. Minarik, M., Sekanina, L.: Evolution of iterative formulas using Cartesian genetic programming pp. 11–20 (2011)
10. Nishiguchi, M., Fujimoto, Y.: Evolution of recursive programs with multi-niche genetic programming (mngp). In: Evolutionary Computation IEEE World Congress on Computational Intelligence. p. 247252 (1998)
11. Silva, S., Costa, E.: Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories. GPEM 10(2), 141–179 (2009)
12. Sloane, N.J.A.: The On-Line Encyclopedia of Integer Sequences (Aug 2014), http://oeis.org/
13. Smith, D.E.: History of mathematics, vol. 1. Courier Dover Publications (1958)
14. Turner, A.J., Miller, J.F.: Introducing A Cross Platform Open Source Cartesian Genetic Programming Library. GPEM (2014), to Appear
15. Turner, A.J., Miller, J.F.: Cartesian Genetic Programming: Why No Bloat? In: EuroGP'14. pp. 193–204 (2014)
16. Turner, A.J., Miller, J.F.: Recurrent Cartesian Genetic Programming. In: PPSN'14. pp. 476–486 (2014)
17. Vassilev, V.K., Miller, J.F.: The Advantages of Landscape Neutrality in Digital Circuit Evolution. In: Evolvable Systems. pp. 252–263 (2000)
18. Walker, J.A., Miller, J.F.: Predicting prime numbers using Cartesian Genetic Programming. In: Genetic Programming, pp. 205–216 (2007)
19. Walker, J.A., Völk, K., Smith, S.L., Miller, J.F.: Parallel evolution using multi-chromosome cartesian genetic programming. Genetic Programming and Evolvable Machines 10(4), 417–445 (2009)
20. Wilson, G., Heywood, M.: Learning recursive programs with cooperative coevolution of genetic code mapping and genotype. In: GECCO'07. pp. 1053–1061 (2007)