Noname manuscript No.
(will be inserted by the editor)

Introducing A Cross Platform Cartesian Genetic Programming Library

Andrew James Turner · Julian Francis Miller

the date of receipt and acceptance should be inserted later

Abstract Cartesian Genetic Programming (CGP) is a form of Genetic Programming (GP) which encodes computational structures as generic cyclic or acyclic graphs. This letter introduces a new cross platform Cartesian Genetic Programming Library intended for use in teaching, academic research and real world applications. This new CGP library is currently capable of evolving symbolic expressions, Boolean logic circuits and Artificial Neural Networks but can easily be extended to other domains. The CGP library, documentation and tutorials are all available at `www.cgplibrary.co.uk`.

Keywords Cartesian Genetic Programming · Software Library · NeuroEvolution

1 Introduction

Cartesian Genetic Programming (CGP) [11] is a form of Genetic Programming (GP) [8] which encodes graph-based computational structures. CGP has had over 15 years of development [9] and has been shown to have a number of advantages over traditional GP [11]. These advantages include: resilience to bloat [10] [19], the presence of neutral genetic drift which has been shown to aid the search process [13] [22] [24], native support for multiple-input multiple-output problems, capable of creating cyclic as well as acyclic graphs [21], and allows for internally calculated values to be reused multiple times [11].

Andrew James Turner
The University of York
Electronics Department
Intelligent Systems Group
York, UK
E-mail: andrew.turner@york.ac.uk

Julian Francis Miller
The University of York
Electronics Department
Intelligent Systems Group
York, UK
E-mail: julian.miller@york.ac.uk

Despite these advantages CGP has not been adopted to the same extent as standard GP [23]. Key necessities for a method or technique becoming widely used are its benefits compared with other methods, ease of use and availability. Currently a number of CGP implementations are available at the CartesianGP homepage [12]. However these implementations are typically under documented, unfriendly to new users and adapting them to new situations requires editing and understanding of the code base.

This letter introduces a new CGP library intended to be of use in teaching, academic research and applications. What distinguishes this implementation from others is that it defines a well documented CGP Application Programming Interface (API); as opposed to a graphical or command line tool. This API provides functionality for high level applications of CGP to a given task, lower level customisation of the CGP algorithm and deploying evolved programs in their intended applications. The library should be thought of as a set of tools for working with CGP. An advantage of using a well defined API is that the user does not need to understand or edit the underlying implementation in order to use the CGP library. Additionally users can benefit from backwards compatible updates to the library. An advantage of creating a compiled library is that it can be used natively by the C and C++ programming languages but also imported into other languages including Python. The CGP library can also be compiled for a wide range of operating systems as it only depend upon standard C libraries. The CGP library also comes with complete documentation and numerous tutorials intended to ease the learning curve.

The rest of the letter is structured as follows. Section 2 describes CGP, Section 3 introduces the new CGP library, Section 4 describes the basic use of the CGP library, Section 5 gives planned future developments of the CGP library and finally Section 6 gives a closing discussion.

2 Cartesian Genetic Programming

CGP [11] [14] is a form of GP [8] which typically evolves acyclic computational structures of nodes (graphs) indexed by their Cartesian coordinates. CGP does not suffer from bloat [10] [19]; a drawback of many GP methods [16]. CGP chromosomes contain non-functioning genes enabling neutral genetic drift during evolution [22] [24]. 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 [13]; which is thought to heighten neutral genetic drift.

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.

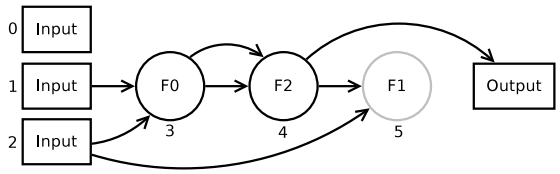

Fig. 1 Example CGP program corresponding to the chromosome: 012 233 124 4

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, the CGP library defines chromosomes 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.

$$F_0C_{0,0}...C_{0,\alpha}F_1C_{1,0}...C_{1,\alpha} \text{ ..... } F_nC_{n,0}...C_{n,\alpha}O_0...O_m \tag{1}$$

An example CGP program is given in Figure 1 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.

3 CGP Library

The CGP library is intended to be used for teaching, academic research and real world applications. This is a very broad scope which is achieved by hiding detail from the user until required, maintaining a well documented API and providing extra tools for specific scenarios.

For instance, in the case of teaching, the CGP library can be applied to a given task with very little "boilerplate" code; see Section 4. Then if required, all of the typical parameters (mutation rate, μ, λ, evolutionary strategy, etc) can be controlled via a simple set functions. Additionally each evolutionary stage (selection scheme, fitness function etc) can be inspected and edited in isolation. All of this is achieved though the API and so details can be hidden or introduced as required.

In the case of academic research, the ability to control evolutionary parameters and implement custom evolutionary stages becomes important. Nearly all of the evolutionary stages used by the CGP library can be redefined to custom versions using the API. Additional functionality is provided to conduct multiple runs to assess average behaviour. The ability to set random number seeds in order to

repeat experiments is provided. Additionally results can be saved to easily parsed comma separated value (.csv) files for storage and further analysis.

In the case of real world applications, extra functions are provided to save, load and execute individual chromosomes. This enables found solutions to be stored distributed and deployed in their intended application. The ability to remove inactive nodes is also provided to reduce the size of saved and loaded chromosomes. This also increases performance as inactive nodes do not have to be processed when evaluating evolved genotypes.

In all cases, accessibility and ease of use are always important. To this end CGP is an open source project available at `www.cgplibrary.co.uk` with the development code hosted with github [15] at `https://github.com/AndrewJamesTurner/CGP-Library`. The ease of use comes from providing a simple API, full documentation and numerous tutorials introducing various aspects of the library with example code.

"Out of the box" the CGP library can be used for symbolic regression, creating Boolean logic circuits and building Artificial Neural Networks. However, the CGP library also allows users to define their own custom node functions; using the API. Therefore the CGP library can be applied to many additional domains. By default the CGP library fitness function is configured for supervised learning tasks, but by implementing custom fitness functions CGP can also be applied to reinforcement learning.

3.1 Visualisation

Visualisation tools are often useful in order to gain an understanding of the solutions found or attempted during evolution. The CGP library currently provides two methods for inspecting chromosomes. The first function, *printChromosome*, displays chromosomes as text in the terminal / command prompt. A typical CGP chromosome displayed using *printChromosome* is given in Table 1. Each input and functioning node is labelled with its index in the chromosome. There is a textual description of the node e.g. *input* for input nodes or the operation for the function nodes. Function node operations are followed by space separated values describing each nodes inputs. Active nodes are also labelled with an '*'. Finally the last line gives the nodes used as chromosome outputs.

The second method for visualising CGP chromosomes makes use of the open source cross platform Graphviz utility [1]. The CGP library function *saveChromosomeDot* creates a Graphviz ".dot" file which can be used by Graphviz to create a image similar to that in Figure 2. The chromosomes are displayed with the inputs on the left, outputs on the right and the position of the internal nodes optimised by Graphviz. Function nodes are labelled with their functionality and given in bold if active.

3.2 NeuroEvolution

NeuroEvolution is a recent application of CGP termed Cartesian Genetic Programming of Artificial Neural Networks (CGPANN) [7] [17]. NeroEvolution is the application of Evolutionary Algorithms to the training of Artificial Neural

Table 1 Example CGP chromosome displayed using *printChromosome*.

(0):	input			
(1):	mul	0	0	*
(2):	mul	1	1	*
(3):	sub	2	1	*
(4):	add	2	3	*
(5):	sub	3	4	*
(6):	div	0	1	
(7):	sub	0	0	
(8):	add	1	5	*
(9):	mul	3	1	*
(10):	sin	2	3	
(11):	sin	6	0	
(12):	div	0	9	
(13):	add	8	9	*
(14):	mul	7	7	
(15):	add	7	3	
outputs:	13			

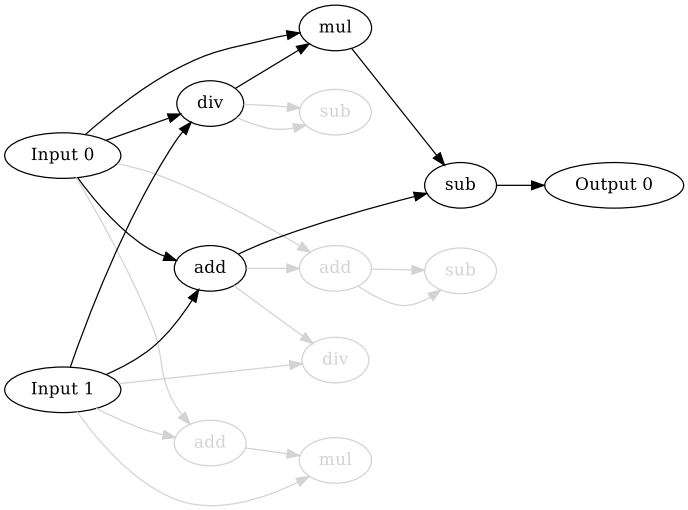

Fig. 2 Example CGP chromosome displayed using *saveChromosomeDot*.

Networks (ANN). CGPANN utilises CGP to evolve the connection weights and topology[18] of ANNs. CGPANN is also capable of evolving both homogeneous and heterogeneous ANNs [20].

The CGP library is capable of using CGP to evolve ANNs by simply using suitable node functions i.e. logistic sigmoid. The necessary connection weights are always present in the CGP library but ignored unless required. The range of the the connections weights and other parameters associated with evolving ANNs can also be set through the API.

3.3 Licenses

CGP library is released under the open source GNU lesser general public license version 3 [3]. The library is released under the *lesser* general public license so it can be used in commercial applications under the conditions given in the license. The documentation associated with the CGP library is released under the open source GNU Free Documentation License version 1.3 [2].

4 Using the CGP Library

Complete documentation for the CGP library, including installation and the API can be found at `www.cgplibrary.co.uk`. In this section only a basic use case is described.

The CGP library uses a number of structures to store data associated with the library; such as the CGP library parameters, training sets and chromosomes. Functions are provided to initialise and free these structures.

A parameters structure is used to store the general parameters which control the evolutionary strategy used by CGP; for instance it describes the selection scheme and mutation method to be used. Parameter structures are initialised using *initialiseParameters* which takes as arguments the dimensions of the chromosomes to be evolved. Many of the default values stored in parameter structures, such as the selection scheme, can be altered but this is not necessary for basic use. Newly initialised parameters structures contain an empty function set and it is the responsibility of the user to populate this function set with functions. This is achieved using *addNodeFunction* which takes as arguments a initialised parameters structure and a comma separated string of function names. There are currently 24 possible node functions separated into three types[1]: symbolic functions, Boolean logic gates and neuron transfer functions. For inspections of the status of the parameters structure *printParameters* displays the parameters in the terminal or command prompt.

The dataSet structures store training or testing data which can be used by the fitness function when assigning fitnesses to chromosomes. DataSet structures can be initialised using *initialiseDataSetFromFile* which takes as arguments the location of the file containing the training/testing data; given in a specific format[2].

Once a parameters structure (with a populated function set) and a dataSet structure has been initialised, CGP can be applied to a given task. This can be achieved using *runCGP* which takes as arguments parameters and dataSet structure as well as the number of generations allowed before terminating the search. After *runCGP* has terminated it returns an initialised chromosome structure containing the best chromosome (solution) found. This chromosome can be visualised using *printChromosome*.

As can be seen, very little "boilerplate" code is required to use the CGP library. Additionally, repeatedly applying CGP to a given task to determine average

[1] Interestingly using a mixture of these three types is also possible.

[2] The first line contains the number of inputs, outputs and the number of data samples. Subsequent lines contain the inputs followed by the outputs for each sample. All values are comma separated.

behaviour requires only slight modification and the use of *repeatCGP* instead of *runCGP*.

Example C code which follows the previous description is given in Listing 1. As can be seen very few lines of C are required to apply CGP to a given task.

Listing 1 Example use of CGP library.

```
#include <stdio.h>
#include <cgp.h>

int main(void){

    struct parameters *params = NULL;
    struct dataSet *trainingData = NULL;
    struct chromosome *chromo = NULL;

    int numInputs = 1;
    int numNodes = 50;
    int numOutputs = 1;
    int nodeArity = 2;

    int numGens = 1000;

    params = initialiseParameters(numInputs, numNodes,
        numOutputs, nodeArity);

    addNodeFunction(params, "add,sub,mul,div");

    printParameters(params);

    trainingData = initialiseDataSetFromFile("temp.data");

    chromo = runCGP(params, trainingData, numGens);

    printChromosome(chromo);

    freeDataSet(trainingData);
    freeChromosome(chromo);
    freeParameters(params);

    return 0;
}
```

5 Future Developments

This section will discuss planned future developments of the CGP library. Additional requests for features can be sent to `andrew.turner@york.ac.uk`.

Additional Node Functions. The CGP library currently contains twelve symbolic functions, seven Boolean logic functions and five neuron transfer functions. This list will be continue to be extended in future releases.

Recurrent CGP. CGP can be used to create cyclic (recurrent) as well as acyclic graphs [21]. Currently the CGP library only creates acyclic programs and will be extended to be also capable of creating cyclic programs. This will allow application to partially observable tasks as well as creating recurrent ANNs.

Custom Mutation Methods. Currently the CGP library provides no functionality to create custom mutation methods through the API. Instead common mutation methods used by CGP are provided by the library. The ability to create custom mutation methods using the API will be included in future releases. For instance, recently Goldman and Punch have suggested highly fficient new mutation operators [5, 4]

Optimisation. The initial goal of the CGP library was to be simple to use and highly extensible. Now this has been achieved, the focus will change to also include speed optimisation; an important aspect of machine learning.

Additional Language Bindings. As the CGP library is compiled into a shared object or dynamic library it can be bound to other languages. The intention is to at least provide bindings for python using the ctypes package [6].

6 Discussion

This letter has introduced a new cross platform open source CGP library intended for teaching, academic research and real world applications. What distinguishes this CGP library from previous implementations is that it defines a well documented API which can be used to apply CGP to many areas. For instance it can be used to apply CGP to a given task, used as the "back-end" to implement other CGP software (such as a CGP command line tool) or used in real applications to implement evolved solutions.

The CGP library also includes full documentation and tutorials. This is included to ease the learning curve for new users and to introduce more advanced features of the library. It is hoped that this new CGP library will encourage others to try CGP as an alternative to traditional GP.

As of now the CGP library is on its first (non-beta) release and is still under active development. For further details, feature requests, bug reports or to contribute please contact `andrew.turner@york.ac.uk`.

References

1. Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz open source graph drawing tools. Graph Drawing (2001)
2. GNU: GNU Free Documentation License (2014). URL `https://www.gnu.org/licenses/fdl.html`
3. GNU: GNU Lesser General Public License (2014). URL `https://www.gnu.org/licenses/lgpl.html`
4. Goldman, B., Punch, W.: Analysis of Cartesian Genetic Programming's Evolutionary Mechanisms. Evolutionary Computation, IEEE Transactions on **PP**(99), 1–1 (2014). DOI 10.1109/TEVC.2014.2324539. In press

5. Goldman, B.W., Punch, W.F.: Reducing wasted evaluations in cartesian genetic. In: Proceedings of the 16th European Conference on Genetic Programming (EuroGP), vol. 7831, pp. 61–72. Springer Verlag (2013)
6. Heller, T.: ctypes (2014). URL `https://pypi.python.org/pypi/ctypes`
7. 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)
8. Koza, J.R.: Genetic Programming: vol. 1, On the programming of computers by means of natural selection, vol. 1. MIT press (1992)
9. Miller, J.F.: An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In: Proceedings of the Genetic and Evolutionary Computation Conference, vol. 2, pp. 1135–1142. Citeseer (1999)
10. Miller, J.F.: What bloat? Cartesian genetic programming on Boolean problems. In: 2001 Genetic and Evolutionary Computation Conference Late Breaking Papers, pp. 295–302 (2001)
11. Miller, J.F. (ed.): Cartesian Genetic Programming. Springer (2011)
12. Miller, J.F.: cartesiangp (2014). URL `http://www.cartesiangp.co.uk/`
13. Miller, J.F., Smith, S.: Redundancy and computational efficiency in Cartesian genetic programming. Evolutionary Computation, IEEE Transactions on **10**(2), 167–174 (2006)
14. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: Proceedings of the Third European Conference on Genetic Programming (EuroGP), vol. 1820, pp. 121–132. Springer-Verlag (2000)
15. Preston-Werner, T., Wanstrath, C., Hyett, P.: github (2014). URL `https://github.com/`
16. 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)
17. Turner, A.J., Miller, J.F.: Cartesian Genetic Programming encoded Artificial Neural Networks: A Comparison using Three Benchmarks. In: Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO-13), pp. 1005–1012 (2013)
18. Turner, A.J., Miller, J.F.: The Importance of Topology Evolution in NeuroEvolution: A Case Study Using Cartesian Genetic Programming of Artificial Neural Networks. In: Research and Development in Intelligent Systems XXX, pp. 213–226. Springer (2013)
19. Turner, A.J., Miller, J.F.: Cartesian Genetic Programming: Why No Bloat? In: Genetic Programming: 17th European Conference, EuroGP-2014 (2014). To appear
20. Turner, A.J., Miller, J.F.: Neuroevolution: The importance of transfer function evolution and heterogeneous networks. In: Proceedings of the 50th Anniversary Convention of the AISB, pp. 158–165 (2014)
21. Turner, A.J., Miller, J.F.: Recurrent Cartesian Genetic Programming. In: 13th International Conference on Parallel Problem Solving from Nature (PPSN 2014) (2014). To appear
22. Vassilev, V.K., Miller, J.F.: The Advantages of Landscape Neutrality in Digital Circuit Evolution. In: Proc. International Conference on Evolvable Systems, *LNCS*, vol. 1801, pp. 252–263. Springer Verlag (2000)
23. White, D.R., McDermott, J., Castelli, M., Manzoni, L., Goldman, B.W., Kronberger, G., Jaśkowski, W., OReilly, U.M., Luke, S.: Better gp benchmarks: community survey results and proposals. Genetic Programming and Evolvable Machines **14**(1), 3–29 (2013)
24. Yu, T., Miller, J.F.: Neutrality and the Evolvability of Boolean function landscape. In: Proc. European Conference on Genetic Programming, *LNCS*, vol. 2038, pp. 204–217. Springer (2001)