A Comparison of Cartesian Genetic Programming and Linear Genetic Programming

Garnett Wilson[1,2] and Wolfgang Banzhaf[1]

[1] Memorial Univeristy of Newfoundland, St. John's, NL, Canada
[2] Verafin, Inc., St. John's, NL, Canada
{gwilson,banzhaf}@cs.mun.ca

Abstract. Two prominent genetic programming approaches are the graph-based Cartesian Genetic Programming (CGP) and Linear Genetic Programming (LGP). Recently, a formal algorithm for constructing a directed acyclic graph (DAG) from a classical LGP instruction sequence has been established. Given graph-based LGP and traditional CGP, this paper investigates the similarities and differences between the two implementations, and establishes that the significant difference between them is each algorithm's means of restricting inter-connectivity of nodes. The work then goes on to compare the performance of two representations each (with varied connectivity) of LGP and CGP to a directed cyclic graph (DCG) GP with no connectivity restrictions on a medical classification and regression benchmark.

Keywords: Linear Genetic Programming, Cartesian Genetic Programming.

1 Introduction

Genetic programming implementations have been proposed that evolve populations of individuals that are constructed as graphs. Two prominent options in the literature that model GP individuals in this way are Cartesian Genetic Programming (CGP) [1-3] and Linear Genetic Programming (LGP) formulated as a graph structure. LGP in graph form was first presented in [4, 5], with the algorithm for the conversion of imperative instructions to graph formally stated in [6]. The goal of this work was to definitively determine the differences and similarities between CGP and LGP. The comparison motivated an obvious new representation to compare connectivity of the implementations: a directed, cyclic graph (DCG) version of CGP, which is subsequently empirically compared to the original LGP and CGP representations on two types of benchmark problems. The DCG alternative in this paper is simply referred to as "DCG" and is the CGP implementation with the input nodes allowing cycles in the graph. That is, there is simply no restriction on the permitted input nodes: the inputs for a given node may refer to other nodes that occur further "ahead" in the graph, or permit the node to reference itself. Many other more or less elaborate DCG implementations have been introduced in the past, often with the aim of relaxing the restriction of using only feed-forward connectivity to adapt the graphs to real world

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 182–193, 2008.

applications. The aim of this paper, rather than to survey graph-based GP approaches, is to investigate the fundamental difference between traditional forms of LGP and CGP, and their restrictions in connectivity.

The following section describes the Cartesian Genetic Programming (CGP) implementation and the components of its representation, with Section 3 describing the implementation and representation of Linear Genetic Programming (in its graph form) in a similar vein. Section 4 compares CGP and LGP implementations to determine their fundamental differences and similarities. Section 5 applies CGP and LGP, each with two parameterizations with differing connectivity constraints, and an unrestricted connectivity DCG, to a classification and regression benchmark.

2 Cartesian Genetic Programming

Cartesian genetic programming (CGP) represents phenotypes of individuals as a grid of nodes addressable in a Cartesian coordinate system. Formally, a Cartesian program is defined by Miller in [3] as the set $\{G, n_i, n_o, n_n, F, n_f, n_r, n_c, l\}$ where G is the genotype that is a set of integers to be described, n_i is the indexed program inputs, n_n is the node input connections for each node, and n_o is program output connections. The set F represents the n_f functions of the nodes, and n_r, n_c are the number of nodes in a row and column, respectively. The levels back parameter, l, indicates how many previous columns of cells have their outputs connected to a node in the current column (with primary inputs treated as node outputs). Program inputs are permitted to connect to any node input, but nodes in the same column are not allowed to be connected to each other. Any node can be either connected or disconnected. See Figure 1 for a diagram of a typical CGP graph.

A graph of the individual consists of a string of integers specifying, firstly, n_n inputs and one internal function for each node, and lastly the n_o program outputs. The CGP genotype thus takes the form of the string of integers

$$C_0, f_0;\ C_1, f_1; \ldots ; C_{cr-1}, f_{cr-1}; O_1,\ O_2,\ \ldots, O_m \tag{1}$$

where C_i indicates the points to which the inputs of the node are connected, and each node is given an associated user-defined function f_i. It is possible to have a list composed of functions with different arities by setting the node arity to be the maximum arity present in the function list and allowing nodes that require fewer inputs to simply ignore the extra inputs. Node 0, described by C_0, f_0, always has an output label that is one greater than the number of program inputs (denoted n in Figure 1). There are also m output genes O_i corresponding to the m program outputs.

In principle, CGP is capable of representing directed multigraphs but has only been used thus far to represent directed acyclic graphs (DAGs). If CGP only encodes DAGs, then the set of possible alleles for C_i are restricted so nodes can only have their inputs connected to either program nodes from a previous (left) column or program inputs. (In other words, they have “feed-forward” connectivity.) As stated by Miller and Smith [2], in many actual CGP implementations the number of rows (r) is set to one, and thus the number of columns (c) is the maximum allowed number of nodes. The levels-back parameter (l) can thus be chosen to be any integer from one to the number of nodes in the graph (n). The output genes are also unnecessary if the program outputs are taken from the m rightmost consecutive nodes when only one row is used. The generic form of CGP is presented in Figure 1 (left), along with typical practical restrictions (right).

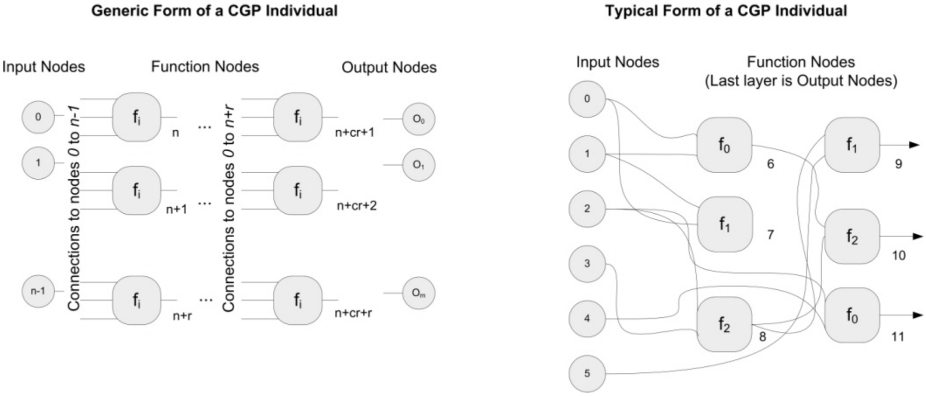

Generic Form of a CGP Individual
Input Nodes
Function Nodes
Output Nodes
0
1
n-1
Connections to nodes 0 to n-1
f_i
n
n+1
n+r
Connections to nodes 0 to n+r
n+cr+1
n+cr+2
n+cr+r
O_0
O_1
O_m
Typical Form of a CGP Individual
Input Nodes
Function Nodes
(Last layer is Output Nodes)
0
1
2
3
4
5
f_0
6
f_1
7
f_2
8
f_1
9
f_2
10
f_0
11

Fig. 1. The generic (left) and typical (right) CGP representations where f_i is a member of the function set, n is the number of inputs, m is the number of outputs, c is the number of columns, and r is the number of rows

To relate CGP in practice to CGP as originally defined in [3], we recall that a CGP program is formally defined by

$$\{G, n_i, n_o, n_n, F, n_f, n_r, n_c, l\} \tag{2}$$

For simplicity, since we are interested in the final graphical representation of a CGP individual, we can eliminate G (the integer representation of the graphical elements n_o, n_n which is redundant for the purposes of representing only the components of a CGP graph) and F (the set of user-defined functions that will be represented as nodes) that are not themselves components of the graph. The CGP graph is now represented

$$\{ n_i, n_o, n_n, n_f, n_r, n_c, l\} \tag{3}$$

In practice, the n_o program outputs need not be used in a graph representation [2]. Instead, recall that some $m < c$ rightmost consecutive nodes for c columns provide outputs when one row is used. This eliminates the need of the n_o variable in the graph representation. Also, when the number of rows is set to one, the number of nodes in a column n_c will always be one and it is necessarily the case that $n_r = c$ (where c is the number of columns = number of internal nodes). Finally, the levels back parameter is often set to be the number of columns, c (as in [2]) to allow a given node to connect to any previous node, but l can be set to any integer k, $k < c$. Making appropriate substitutions, this gives us the typical graph representation of

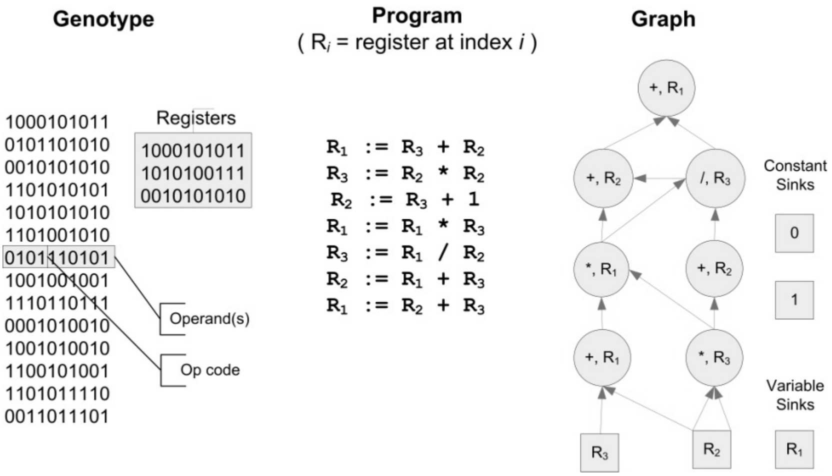
$$\{\, n_i,\ 0,\ n_n,\ n_f,\ 1,\ c,\ l \} \tag{4}$$

3 Graph Representation of Linear Genetic Programming (LGP)

In linear genetic programming (LGP), the genotype individuals have the form of a linear list of instructions as a binary string [6]. This binary string may in turn be interpreted or represented as a set of integers, just as in CGP genotype representations. Program execution is that of a simple register machine (Von Neumann computer), and instructions are made up of opcodes and operands (providing linear forms of Functional and Terminal sets, respectively). As the program executes, it alters the contents of the internal registers (or stack) and solution register(s).

When the bit strings are interpreted, they correspond to members of the Functional (and sometimes Terminal) sets to produce a phenotype solution. For instance, the binary sequence "011" in the individual's genotype could be interpreted as the functional set member "addition" in the phenotype. The immediately following bits often refer to destination and source registers, if registers are used as opposed to a stack. The phenotype is then evaluated to determine the corresponding fitness. The structure of a linear GP individual is depicted below in Figure 3.

The instruction sequence (imperative) view of a linear program can be transformed into an equivalent functional representation in the form of a directed acyclic graph (DAG). This is simply an alternate way of representing the linear program and registers. The directed nature of the graph better enables the deciphering of functional dependencies and execution flow during interpretation of the instructions. For details of the formal algorithm to convert LGP to a DAG, the reader is referred to [6]. The application of the algorithm to imperative instructions produces a DAG such that the number of inner nodes always equals the number of imperative instructions. Each of these inner nodes includes an operator, and has as many incoming edges as there are operands for that operator in the corresponding imperative instruction. Sink nodes have no outgoing edges and are labeled as registers or constants. While the nodes in [6] are labeled with only operators, the nodes are plotted as unique nodes in virtue of target register and operator at a particular execution point. The maximum number of sink nodes is thus the total number of registers and constants in the terminal set. Upon completion of the DAG, the sinks represent input variables of the program. Constant sinks and inputs may be pointed to from every program position. An LGP program in the form of binary genotype, interpreted program, and graph structure is shown in Figure 2 below.



```
R1  := R3 + R2
R3  := R2 * R2
R2  := R3 + 1
R1  := R1 * R3
R3  := R1 / R2
R2  := R1 + R3
R1  := R2 + R3
```

Fig. 2. Representations of a LGP individual

4 Comparison of CGP and LGP Representations

We must now determine whether or not LGP and CGP graphs are similar representations, and if they are, what is the nature of that similarity? In the case of both graphs, each unique node is identified by a function and the nodes from which input to the function is received. Again, in both cases, inputs for a given inner node can only be received from nodes or variable/constant sinks (inputs for CGP) that have already been established in the graph. The semantic representation of the nodes is thus relevantly similar between the two representations.

Using the seven-tuple $\{n_i, n_o, n_n, n_f, n_r, n_c, l\}$ representation of CGP, how would LGP be formulated? In LGP, the constant and variable sinks are effectively program inputs, n_i. By specifying the output to be the content of an LGP register at the finish of imperative instruction execution, n_o is set to 0 as in common CGP practice. That is, there are no special output nodes in either CGP or LGP. There are a given number of internal nodes in LGP specified by input connections and functions; let us specify these as n_n just as in CGP implementations. There is also a function set consisting of n_f functions. As in the most common CGP implementations, there is no notion of separate rows and columns in LGP, so there are $n_c = c$ and $n_r = 1$ nodes in an LGP graph. If nodes are capable of being connected to any previous node in LGP (as is typical in CGP), then $l = c$. However, the usual in LGP (as presented in [6]) is that a given node can only connect to the particular nodes in the previous layers *that last used the registers specified by the function in the current node.*

Given the considerations thus far in this section, the tuple representing a LGP graph contains the same representative elements as typical CGP (Eq. 4), with the exception that nodes in LGP graphs take input from previous nodes that last used the registers the current node requires as inputs. Thus, the levels-back parameter (l) of the tuple is not relevant to LGP graphs, giving the tuple where l is not applicable (*n/a*):

$$\{\ n_i,\ 0,\ n_n,\ n_f,\ 1,\ c,\ \text{n/a}\} \quad (5)$$

The representation elements of typical LGP and typical CGP in practice really only differ in the parameter of the number of levels back. (Compare Eq. 4 and Eq. 5.) In a practical sense, this means that there are different restrictions on how a given inner (function) node in the two implementations can refer to incoming nodes. The interconnectivity of the LGP graph is thus constrained in an implicit way, as opposed explicitly specifying the levels-back parameter in CGP.

In terms of genotype representation, a CGP genotype is a series of pairs representing nodes. Each pair consists of a set of points to which the inputs of the node are connected, and the function for the node. Upon listing the connections and functions for all nodes, the nodes from which the output(s) are to be taken are specified in the genotype (see Section 2), or they may simply be specified as some number of last nodes in the graph as a parameter outside the genotype. This is largely a design decision, but the specification of output nodes may or may not be under control of evolution as part of the genotype in CGP. In contrast, an LGP implementtation typically chooses particular register(s) in which the output is to be found, and the output registers are not listed in the genome. As mentioned previously, the output nodes are typically taken to be the last layer of graph nodes in CGP, so output nodes are effectively left out of the CGP genotype in modern representations (making the genotype similar to LGP in that the nodes specified for output are not part of the genotype). In the case of both LGP and CGP, one can also have a number of outputs from the registers or nodes, respectively.

An LGP individual's genotype is a list of imperative instructions. Each line represents a function and has some associated registers and a destination register. Using the algorithm of Brameier and Banzhaf, though, the genotype representation can be converted to a graph, which can alternately be described as a listing of nodes including function and input edges. A node is made unique in virtue of three components: source register(s) (or source data) and destination register, function, and when it is executed in the program (placed in the graph). The source register(s) or source data effectively indicate nodes to which the incoming edges are connected because the last nodes having used the source registers of the current node as their target register (or simply the specified input data from variable or sink nodes) will be connected to the incoming edges. The destination register serves to reference the output edge of the node because the next future nodes to reference the current node's destination register as source registers will form an incoming edge from the current node. The final value in the register(s) of interest at the end of execution in LGP are the output value(s) in LGP, and they are the last instances of the nodes labeled with the relevant registers in an LGP graph. Note that this has the same effect as choosing a particular node (or nodes) as the output in CGP, which is what is done typically in current implementations. Also, nodes are actually labeled with only the instruction operator, and the target register can be added to the node label for clarity but is generally used as a temporary variable to plot the LGP graph (see [6]). In this work, the target registers are included on the node labels for clarity of interpretation, but they are generally left out the of the final plot (as in [6]).

Functions are encoded in the same way in both CGP and graph LGP. Collectively, the components of a single instruction in an LGP genotype correspond to a node that is a unique imperative instruction. The only difference in the encoding of the LGP and CGP graph is that there is an explicit identification of the node for future outgoing edges that is encoded in the genome in LGP (in virtue of the instruction's target register), whereas in CGP the nodes are just sequentially ordered as they appear and not encoded as part of the genome. This means that the encoding of the genome restricts what previous nodes get connected to a node in a current layer in graph LGP. In contrast, in CGP that restriction is handled by specifying the levels back (l) parameter, and it is not explicitly coded in the genome. Thus, in LGP the connectivity of the nodes is under evolutionary control since it is part of the genome, but in CGP it is specified *a priori* as a design parameter.

The characteristics of the elements required for graph representation and the genotype structures of CGP and LGP dictate their graphs will be similar (Eq. 4 and Eq. 5). Consider the common CGP using one row and the LGP graph for programs of single non-conditional, non-branching imperative instructions. Typically, these graphs will both involve two inputs per node if Boolean functions such as AND, NAND, OR, and NOR as are typically used in circuit board design are used. However, in both CGP and LGP graphs, the nodes may accept varying numbers of input edges depending on the maximum required by the function with the most arguments in the function set. Furthermore, both graphs are directed, with data only flowing in the direction from input nodes/sinks to output nodes. In other words, programs are restricted such that nodes only have their inputs connected to the program inputs or nodes from a previous column; edges only point in the general direction of the output. To summarize, both CGP and DGP are represented as DAGs with each node capable of any number of input edges. The only difference between CGP and DGP graphs is the restriction on how the input edges are assigned to a node (as discussed in the previous section). Thus, given an unlabelled DAG with arbitrary node layout generated by either GP variant, the user could not readily distinguish between the two without further information, namely the design parameterization of the CGP tuple $\{n_i, n_o, n_n, n_f, n_r, n_c, l\}$ and the number of registers used in the LGP algorithm. See Table 1 below for a summary of the comparison of LGP and CGP graph representations.

Table 1. Comparison of representation components of CGP and LGP (differences in bold)

	CGP Graph	Graph LGP
Tuple	$\{ n_i, 0, n_n, n_f, 1, c, \boldsymbol{l}\}$	$\{ n_i, 0, n_n, n_f, 1, c,$ **n/a**$\}$
Genotype	Integer or binary string	Integer or binary string
Graph Type	DAG	DAG
Node content	Function	Function
Connectivity	**Restricted by levels-back (not under evolutionary control)**	**Restricted by usage of target registers (evolutionary control)**
Incoming edges	Maximum required by function set	Maximum required by function set

5 Comparison of Graph-Based Genetic Programming Techniques on Classification and Regression Benchmarks

In previous sections, the representation elements and their implications for the functionality of the CGP and LGP graph types was discussed. The main difference between LGP and CGP was the mechanism used to restrict the allowed input edges to a given node, including whether or not the edges are under evolutionary control. To provide contrast to, and determine the practical value of, the connectivity restrictions of CGP and LGP, we introduce a new graph type called simply "DCG" for "directed cyclic graph." This new graph type follows the CGP representation, only that each node can accept inputs from any node in the graph. This means that there is no restriction of data flow to only feed-forward connectivity, cycles are permitted, and the levels back parameter is not relevant. LGP graphs can also permit cycles, but the corresponding imperative LGP programs would have to involve jump statements. Such considerations are beyond the scope of this work, as the LGP would no longer conform to the current formal algorithm in [6] which is used here for the comparison of traditional CGP and graph LGP.

Two implementations of CGP with varying connectivity are tried, with levels back being equal to the number of columns (nodes), or only 2. In LGP, two progressively constricting instruction forms are tried: 1 input and 2 input. In the single input implementation, an instruction applies a function to data from a source register *X* and target register *Y*, replacing the data in that same target register *Y*. In the two input implementation, an instruction applies a function to data from two source registers *X* and *Y*, placing the result in another target register *Z*. In addition, the number of inner nodes in LGP graphs is determined by the nature of the instructions: Due to the use of registers in LGP, functions in nodes may draw their input(s) from registers or fitness cases. If there is a larger number of fitness cases than registers (as in the classification benchmark), fewer bits are needed to specify one of four registers, but more bits are needed to load from one of the fitness case fields. In the regression benchmark, there are fewer fitness case features than number of registers. Whether or not to load from register or fitness case is determined by a binary flag. Only the required number of bits is used to interpret a given instruction, resulting in individual-dependent graph sizes. The summary of the general parameterization of the implementations is given in Table 2.

Table 2. General parameterization of CGP, LGP, and DCG implementations

Tournament Style	Steady State, 4 individuals per round
Population size	25
Genotype structure	240 bit string, 4 registers (LGP)
Graph structure	16 inner nodes + input nodes (CGP & DGP), determined by bit string (LGP)
Genotype mutation	point mutation, threshold = 0.9

Here we compare the graph GPs' empirical performance on a real world classification benchmark, namely the Heart Disease data that is part of the UCI Machine Learning Repository [7], and the Mexican Hat regression benchmark as described in [6]. Only mutation is used in these experiments, as crossover in CGP does not make intuitive sense. Furthermore, mutation is appropriately restricted in CGP so that a given node can only refer to previous nodes in the graph according to the levels back parameter. Naturally, mutation is unrestricted in LGP and DGP. Execution is carried out from inner start node to end node in CGP and DCG (execution sequence is already determined by order of instructions in LGP). To allow data (other than default) placed in a node to be fed back through the network in DCG, multiple execution iterations over the inner nodes is required. In DCG experiments, five iterations of the inner nodes are executed per fitness case.

The medical database contains 303 instances (164 negative, 139 positive), each consisting of 13 attributes, with a 14th indicating positive or negative diagnosis. Prior to trials, unknown values were replaced by the mean value of the relevant attribute and the positive or negative diagnosis was changed to '1' or '0', respectively. The results use four-fold cross-validation to verify accuracy of the findings. Each partition consisted of a unique 25% test set and 75% training set and retained the class distribution of the entire data set. If the output of an individual was less than 0 on a fitness case, the case was classified as a negative diagnosis; otherwise the individual classified the case as positive. The function set used was { +, *, -, /, SIN, COS, EXP, NATLOG }, protected as appropriate. Fitness was defined simply as number of correct classifications, and training was conducted for 30 000 rounds. The results are shown in Figure 2. The median and spread shown in the boxplot correspond to the mean accuracy across the four unique test sets used in four-fold cross-validation over the 50 trials. Each box indicates the lower quartile, median, and upper quartile values. If the notches of two boxes do not overlap, the medians of the two groups differ at the 0.95 confidence interval. Points represent outliers to whiskers of 1.5 times the interquartile range. A customized version of the popular Java-based Prefuse [8] framework was created a provide a means of visualizing the final graph topologies, where the best trial in each implementation for the first partition are shown in Figure 3.

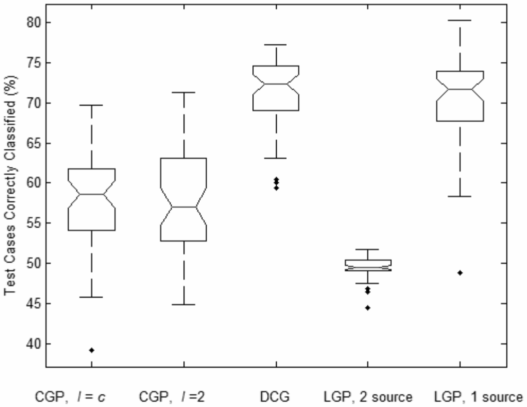


Fig. 3. Boxplot of mean classification accuracy for the Cleveland Heart data set over 50 trials using four-fold cross-validation. Each partition was 75% training, 25% test.

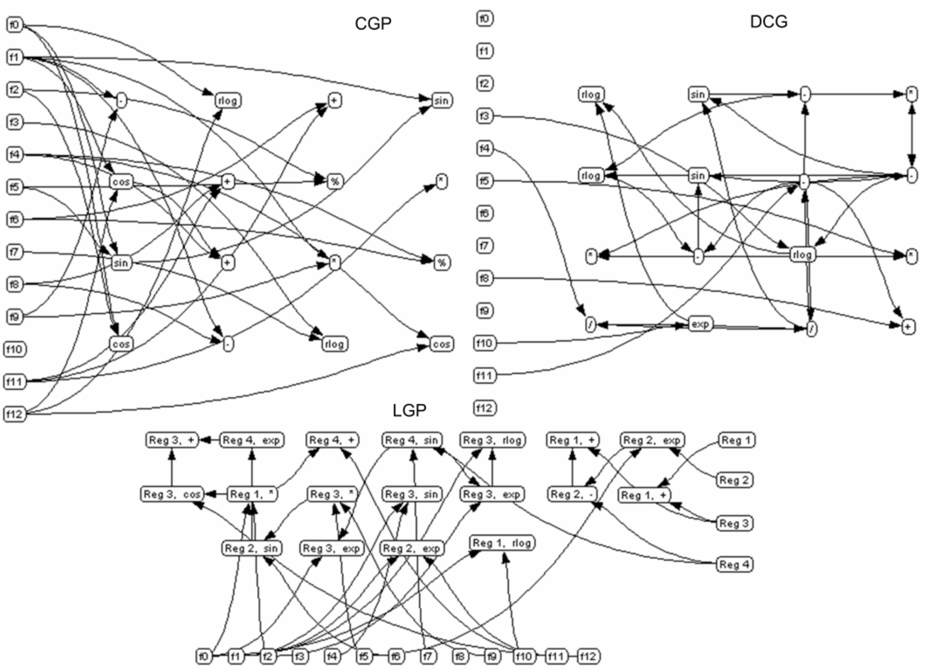


Fig. 4. Individuals corresponding to the best final solution for CGP (*l* = *c*), LGP (2 input), and DCG types for the best trial in a particular partition of the UCI Machine Learning Repository Heart Disease test set. The node corresponding to final classification is the lower right-hand node in the CGP and DCG graphs, and the upper left node in LGP.

Figure 2 indicates that the chosen DCG model (center) outperformed the more restrictive and traditional Cartesian GP implementations (two leftmost). DCG was not outperformed by the more restrictive form of LGP (rightmost), with no statistically significant difference shown (note overlapping notches.) The least restrictive LGP implementation did not perform as well as the other implementations (second from the right). The additional freedom of data flow within the DCG graph due to the admission of cycles enhanced classification ability. Furthermore, in all cases, the restriction of information flow within implementations of both CGP and LGP models led to decreased classification accuracy. In figure 3, the directed edges in the DCG solution show that it clearly takes advantage of its freedom of connectivity and admission of cycles.

The two- Mexican Hat problem as described in [6] is tested on the implementations to provide a regression benchmark. The problem is named for the shape of the three-dimensional plot of its function

$$f_{mexicanhat(x,y)} = (1 - \frac{x^2}{4} - \frac{y^2}{4}) \times e^{(-\frac{x^2}{8} - \frac{y^2}{8})} \qquad (6)$$

Following the parameterization of [6], 400 fitness cases were used, with the input range restricted to [-4.0, 4.0]. The function set consisted of {+, -, x, /, x^y}, protected

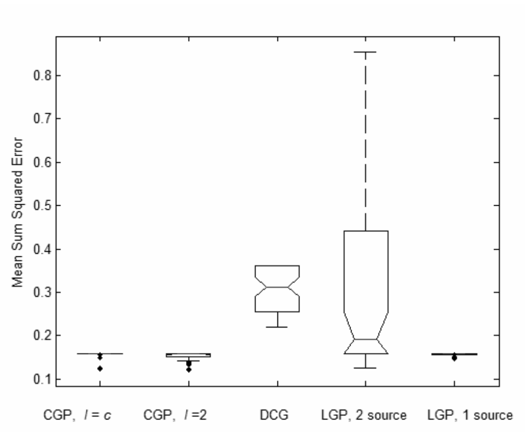


Fig. 5. Boxplot of mean sum squared error for the Mexican Hat problem set over 50 trials

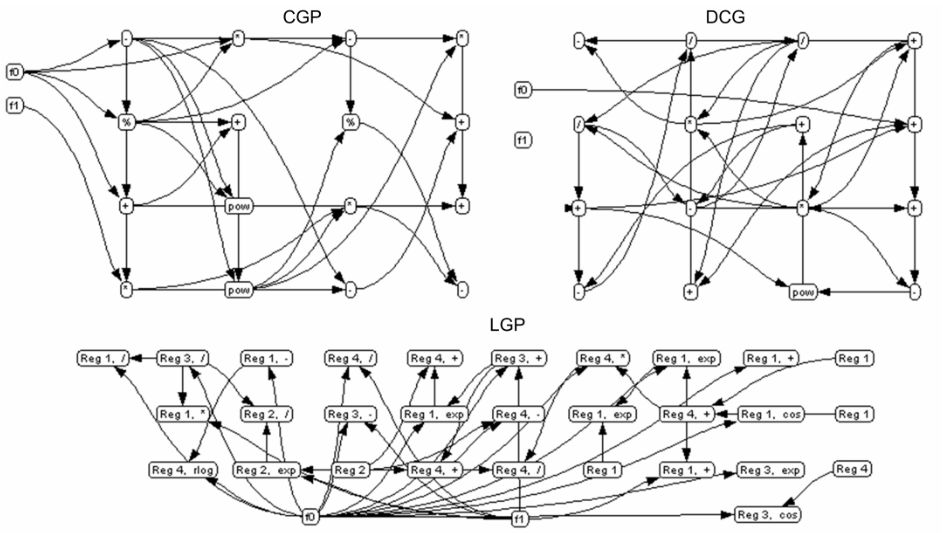


Fig. 6. Individuals corresponding to the best final solution for CGP ($l = c$), LGP (2 input), and DCG types for the best trial for the Mexican Hat problem. The node corresponding to final classification is the lower right node in CGP and DCG, and the upper left node in LGP.

when needed, and tournaments ran for 1000 rounds. Figure 4 shows the boxplot of the mean sum squared errors of the implementations for the Mexican Hat problem, with best final networks of CGP, DCG, and LGP shown in Figure 5.

Given the regression benchmark (Figure 4), DCG clearly does not perform as well. It is also noteworthy that the unrestricted LGP has the greatest variability over all solutions. Comparing classification and regression benchmark performance (Figures 2 and 4), it is evident that less restricted connectivity (DCG and 2 source LGP) is not of benefit in this regression benchmark. Furthermore, all CGP variants and the restricted LGP variant perform best on the regression benchmark but worse on the classification. This difference may be due to the freely connected nature of DCG allowing it to provide a more highly adapted configuration for classification of a complex problem. In contrast, the definite answer in the regression problem, to be found within a lower number of tournament rounds, is hindered by the greater availability of configurations and cycles in the DCG. In Figure 5, we see that this is the case where the best DCG incorporates extensive connectivity while processing only one input, whereas CGP does not maximize its available levels-back flexibility.

6 Conclusions and Future Work

This work establishes that the difference between graph-based LGP and CGP is the means with which they restrict the feed-forward connectivity of their DAG graphs. In particular, CGP restricts connectivity based on the levels-back parameter while LGP's connectivity is implicit and is under evolutionary control as a component of the genotype. Unrestricted forms of LGP and CGP, and DCG, performed well on the real world medical classification benchmark, but the flexibility of the less restricted graph types did not allow them to perform as well on a regression benchmark. In future work, we plan to explore GP-based search using DCGs in an industry-based real world application. Possibilities for future investigation also include a DCG analogy of LGP graphs, and a closer examination of the relationship between performance of the representations and their connectivity characteristics and evolvability.

Acknowledgements. The authors gratefully acknowledge the support of a PRECARN Postdoctoral Fellowship.

References

1. Miller, J.F., Job, D., Vassilev, V.K.: Principles in the Evolutionary Design of Digital Circuits - Part 1. Genetic Programming and Evolvable Machines 1, 8–35 (2000)
2. Miller, J.F., Smith, S.L.: Redundancy and Computational Efficiency in Cartesian Genetic Programming. IEEE Transactions on Evolutionary Computation 10, 167–174 (2006)
3. 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)
4. Banzhaf, W., Nordin, P., Keller, R., Francone, F.: Genetic Programming - An Introduction: On the Automatic Evolution of Computer Programs and Its Applications. Morgan Kaufmann Publishers, San Francisco (1998)
5. Nordin, P.: Evolutionary Program Induction of Binary Mchine Code and its Application. Krehl Verlag, Munster (1997)
6. Brameier, M., Banzhaf, W.: Linear Genetic Programming. Springer, New York (2007)
7. Newman, D.J., Hettich, S., Blake, C.L., Merz, C.J.: UCI Repository of Machine Learning Databases. University of California, Department of Information and Computer Science, http://www.ics.uci.edu/~mlearn/MLRepository.html
8. Heer, J.: Prefuse Interactive Information Visualization Toolkit, http://prefuse.org