Embedded Cartesian Genetic Programming and the Lawnmower and Hierarchical-if-and-only-if Problems

Genetic Programming Track

James Alfred Walker
jaw500@ohm.york.ac.uk

Julian Francis Miller
jfm7@ohm.york.ac.uk

Intelligent Systems Group, Department of Electronics
University of York, Heslington, York, YO10 5DD, UK

ABSTRACT

Embedded Cartesian Genetic Programming (ECGP) is an extension of the directed graph based Cartesian Genetic Programming (CGP), which is capable of automatically acquiring, evolving and re-using partial solutions in the form of modules. In this paper, we apply for the first time, CGP and ECGP to the well known Lawnmower problem and to the Hierarchical-if-and-Only-if problem. The latter is normally associated with Genetic Algorithms. Computational effort figures are calculated from the results of both CGP and ECGP and our results compare favourably with other techniques.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program synthesis*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search

General Terms

Algorithms, Design, Performance

Keywords

Cartesian Genetic Programming, Embedded Cartesian Genetic Programming, Module Acquisition, Automatically Defined Functions, Evolution, Lawnmower problem, Hierarchical-if-and-only-if

1. INTRODUCTION

Embedded Cartesian Genetic Programming (ECGP) is an extension of the directed graph-based Cartesian Genetic Programming (CGP), incorporating ideas from a technique known as Module Acquisition [1]. This allows the automatic acquisition, evolution and re-use of partial solutions in the form of modules. Previous work [15, 17, 16] has shown ECGP to be more computational efficient than CGP on evolving solutions to a range of digital circuit problems (such as even parity, adders and multipliers) and that the speedup grows with problem difficulty. This suggests that ECGP may be even more computational efficient than CGP on harder problems. The aim for this paper is to build on the results from the previous work on ECGP, by applying the technique to two problems that are not related to circuit evolution. The two problems chosen are: the lawnmower problem, which is a well known problem in the GP community and the Hierarchical if-and-only-if (H-IFF) problem, which is normally associated with Genetic Algorithms.

The plan for the paper is as follows: Section 2 is an overview of related work. In section 3 we describe ECGP and compare it with CGP. The details of our experiments, including descriptions of the lawnmower and H-IFF problems are shown in section 4 followed by the results and comparisons for both experiments in section 5. Section 6 gives conclusions and some suggestions for future work.

2. MODULE ACQUISITION, AUTOMATICALLY DEFINED FUNCTIONS AND M-ACROS

Module Acquisition [1] adds two operators to the evolutionary process, *compress* that selects a section of the genotype to make it immune to manipulation from operators (the module) and *expand* which decompresses a module in the genotype therefore allowing this section of the genotype to be manipulated once more. The fitness of a genotype is unaffected by these operators. Module Acquisition allows the possibility of having modules within modules. These techniques have been shown to decrease the time taken to find a solution. Rosca's method of Adaptive Representation through Learning (ARL) [11] also extracted program segments that were encapsulated and used to augment the GP function set. However, Dessi et al [3] showed that random selection of program sub-code for re-use is more effective than Roscas method across a range of problems. Once the contents of modules are themselves allowed to evolve (as in ECGP) they become a form of Automatically Defined Function, however in contrast to Koza's form of Automatically Defined Functions [6] and Spector's Automatically Defined Macros [13], there is no explicit specification of the number of or the internal structure of such modules. This freedom does exist in Spector's PushGP [14].

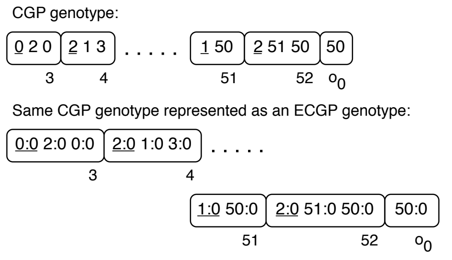

Figure 1: Examples of evolved CGP and ECGP genotypes for H-IFF problem with an 8-bit string (3 inputs, 1 output). For each node, the underlined gene encodes the function and the node type (only the function in CGP). The remaining genes encode the node inputs. Every node encoded in the CGP genotype represents a single output primitive function, therefore every node encoded in the ECGP genotype is of node type 0, and the second integer of each pair encoding the node inputs is always 0. The node index is underneath each node.

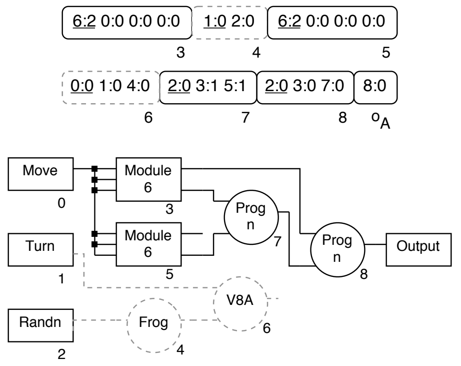

Figure 2: An ECGP genotype and corresponding phenotype for the 8-bit H-IFF problem. The underlined genes encode the function and node type of each node. The function lookup table is: *v8a*(0), *frog*(1), *progn*(2). See section 4.2 for details. The index labels are shown underneath each program input and node in the genotype and phenotype. Module 6 represents a possible structure for a subroutine constructed from the function set. The inactive areas of the genotype and phenotype are shown in grey dashes (nodes 4 and 6).

3. EMBEDDED CARTESIAN GENETIC PROGRAMMING (ECGP)

3.1 Representation

ECGP and CGP share the same structure and represent a program as a directed graph (that for feed-forward functions is acyclic). The benefit of this type of representation is the implicit re-use of nodes in the directed graph.

The genotype is a list of integers that encode the connections and functions of each node of the directed graph. CGP used a program topology defined by a rectangular grid of nodes with a user defined number of rows and columns. However, later work in CGP always chose the number of rows to be one, thus giving a one-dimensional topology. This is always used in ECGP. In CGP, the genotype is a fixed length representation (in terms of nodes and genes). However, in ECGP the *genotype* is a variable length representation (in terms of nodes and genes), in which the number of nodes and genes in the graph can vary but is bounded. The number of nodes in the ECGP genotype vary as a result of the compression and expansion of modules. Also, the number of genes vary as a result of the re-use of modules elsewhere in the genotype, and the module mutation operators changing the number of node inputs. Despite the differences, both genotypes decode into a bounded variable-length directed graph (phenotype), as not all of the nodes encoded in the genotype have to be connected. This allows areas of the genotype to be inactive and have no influence on the phenotype, leading to a neutral effect on genotype fitness called neutrality. This unique type of neutrality has been investigated in detail [7] and found to be extremely beneficial to the evolutionary process on the problems studied. In Figure 1 an example of the differences between a CGP and an ECGP genotype are shown. All of the ECGP genotypes in the initial population have the same number of nodes and genes, and every node represents a primitive function (no modules are present).

Each of the nodes consist of a number of genes. In ECGP, each gene consists of a pair of integers, as opposed to a single integer in CGP. The first integer pair of each ECGP node encodes the primitive function (as in CGP) or module (by their unique identifier) that the node represents and the node type (introduced in ECGP). Node types allow the identification of nodes encoded in the genotype which represent: primitive functions (node type 0), modules that contain an original section of the genotype (node type I) and modules that contain a re-used section of the genotype (node type II). Different node types need to be identified, as operators act differently on the nodes encoded in the genotype depending on their node type (further details are explained in section 3.3). The remaining integer pairs in each ECGP node encode the node inputs. The first integer in each pair encodes the node index in the genotype or program input (terminal), whilst the second integer encodes the output of the node (nodes can have multiple outputs in ECGP). The number of inputs and outputs that a node has is dictated by the arity of its function.

The nodes take their inputs in a feed forward manner from either the output of a previous node or from a program inputs (terminals). The program inputs are labelled from 0 to n-1 where n is the number of program inputs. The nodes in the genotype are also labelled sequentially starting from n to n+m-1 where m is the user-determined upper bound of the number of nodes. If the problem requires k program outputs then k integers are added to the end of the genotype, each one encoding a pointer to the output of a node in the graph where the program output is taken from. These k integers are initially set as pointers to the outputs of the last k nodes in the genotype. Figure 2 shows an ECGP genotype and its corresponding phenotype (a 8-bit H-IFF problem), whilst Figure 3 illustrates the decoding process of the genotype.

Both CGP and ECGP use a (1 + 4) evolutionary strategy as defined below:

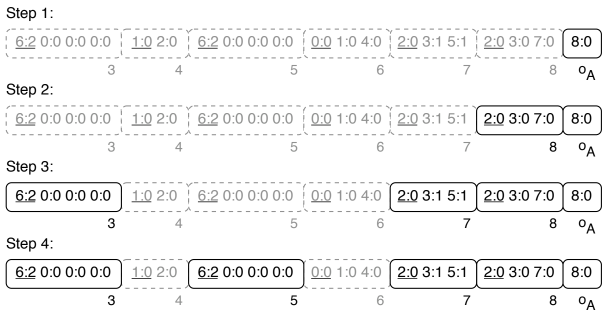

Figure 3: Decoding the ECGP genotype from Figure 2. Step 1: Output A (o_A) connects to the output of node 8, move to node 8. Step 2: Node 8 connects to the output of nodes 3 and 7, move to nodes 3 and 7. Step 3: Nodes 3 and 7 connect to the output of nodes 3 and 5, and program input 0, move to node 5 (as node 3 has already been processed). Step 4: Node 5 only connects to program input 0, therefore the genotype is now decoded.

1. Randomly generate an initial population of 5 genotypes and select the fittest.

2. Carry out point-wise mutation on the winning parent to generate 4 offspring.

3. Construct a new generation with the winner and its offspring.

4. Select a winner from the current population using the following rules:

   (a) If any offspring has a better fitness; the best becomes the winner.

   (b) Otherwise, an offspring with the same fitness as the best is randomly selected.

   (c) Otherwise, the parent remains as the winner.

5. Go to step 2 unless the maximum number of generations is reached or a solution is found.

3.2 Module Representation

A module is represented as a bounded variable length genotype that has the same characteristics of a ECGP genotype. The module genotype consists of a list of integers and is split into two parts: the module header and the module body. The module header contains four integers and stores information about the module. Each of the four integers encodes the module identifier, the number of module inputs, the number of nodes contained in the module and the number of module outputs respectively. The module body encodes the connections and functions of the nodes contained in the module and the module outputs (similar to program outputs), in the same way as any ECGP genotype. An example of a module genotype showing the separate components is shown in Figure 4, where the first block of the module genotype consisting of four numbers represents the module header. The nodes are represented in the same manner as ECGP and the module outputs encode which nodes in the module the outputs are taken from.

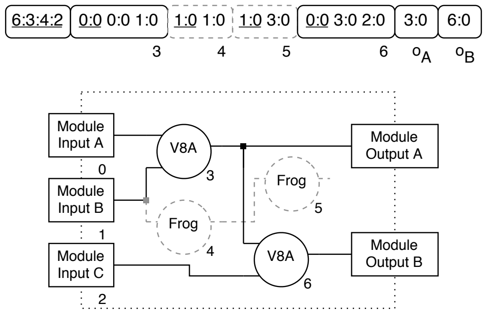

Figure 4: The genotype and corresponding phenotype of a module 6 from Figure 2. The first section of the genotype is the module header. For each node, the underlined genes encode the function, the remaining genes encode the node inputs. The function lookup table is: *v8a*(0), *frog*(1), *progn*(2). The index labels are shown underneath each module input and node in the genotype and phenotype.The inactive areas of the genotype and phenotype are shown in grey dashes. The dotted box represents the edges of the module.

The size of a module genotype is determined by the number of nodes and module outputs that it encodes. The number of nodes encoded in the module genotype is bounded between a minimum limit of two (any fewer and it would either be an empty module or a primitive function) and a maximum limit that is set by the user. Likewise, the number of module outputs encoded in the module genotype is also bounded between a minimum limit of one (otherwise there would be no way to connect to the module and access its result to the given inputs) and a maximum of p module outputs, where p is equal to the number of nodes contained in the module (one module output per node). The number of module inputs that a module is allowed to have is also restricted between a minimum of two and a maximum of $2p$ module inputs. However, the number of module inputs allowed does not affect the size of the module genotype, as they are not encoded in the module genotype. In its current form, ECGP only allows modules to contain nodes representing primitive functions rather than nodes representing other modules.

Once a module is created, the module genotype is stored in the module list, which is an extension of the primitive function list. This allows any node in the genotype of an individual to be mutated into any module or primitive function present in either of these lists for that generation. The module list is dynamic and has no restrictions on its maximum size and is updated every generation when the fittest individual (chosen in accordance with the evolutionary strategy used in Section 3.1) in the generation is promoted to the next generation (i.e. the next generation inherits the module list of the fittest individual in the previous generation). This creates a regulatory control of the size of the module list, so that it does not grow beyond a certain size. The nodes contained inside the module are not necessarily connected and are immune from the main genotype point mutation operator. However, the module itself is allowed to be mutated by the module mutation operators (see Section 3.3).

3.3 Operators

ECGP extends CGP by allowing the use of dynamic acquisition, evolution and the re-use of modules. This is achieved through extra mutation operators, which are used in conjunction with the genotype point mutation of CGP. The compress operator constructs modules by selecting two random points in the genotype (in accordance with the rules for the module size restrictions) and encapsulates all the nodes (of type 0) between these two points into a new module, which is encoded into a module genotype as described earlier. Note, that if there any nodes of type I or II between the two selected points, the compress operator does not take place (this is because at present we do not allow modules within modules). The number of module inputs a module is initialized with, is determined by the number of connections between the inputs of the nodes that are going to be encapsulated into a module, and the outputs of any previous nodes or program inputs (terminals) in the genotype, when the module is created. Likewise, the number of module outputs possessed by a module is determined by the number of connections between the inputs of the latter nodes in the genotype and the outputs of the nodes that are going to be encapsulated in the module, when it is created. Any module created by the compress operator is represented in the genotype as a type I node. The gene representing the function and node type in any type I node is immune from the genotype point mutation operator therefore allowing the type I node to remain in the genotype of an individual until it is removed by the expand operator.

The expand operator destroys a type I node by replacing it in the genotype with the nodes contained in the module, which the type I node represented. The inputs of the latter nodes in the genotype are updated in the final stage of both the compress and expand operators, so all the connections remain intact. The compress and expand operators only make a structural change to the genotype and have no affect on genotype fitness, as the genotype before and after the action of these operators represent the same directed graph. The expand operator has twice the probability of being applied to the genotype than the compress operator. We found that this introduces a pressure for good modules to replicate quickly in the genotype of an individual in order to survive. This can be seen as survival-of-the-fittest modules within the genotype itself.

Modules can replicate within a genotype through the action of the genotype point mutation operator. This is identical to the operator used in CGP, except it can mutate the function of a node (of type 0 or II) to any of the primitive functions or any available modules in the module list. If a node is mutated to represent a module, it is classed as a type II node. The genotype point mutation operator can also mutate the function of a type II node to any of the predefined functions or any available modules in the module list. It can also mutate any of the inputs of a type II node in the same way it would mutate the inputs of a type 0 node. If the function of a type 0 or type II node is mutated, the new node keeps however many of the original nodes inputs it needs, and randomly generates any extra inputs it requires. Type II nodes are also immune from the expand operator, as this could cause excessive growth of the genotype that could possibly lead to bloat.

To summarize the properties of node types 0, I and II are shown in Table 1. A module can be represented by two node types (node type I and II) in order to reduce the excessive growth of the genotype and to induce a selection pressure on the modules. Therefore, the modules have to replicate in the genotype (make the transition from being represented by a type I to a type II node) and be associated with a high fitness genotype in order to survive. Once the module is represented by a type II node it is harder for the module to be removed from the module list, as it has a lower probability of being removed from the genotype (as it cannot be expanded). This is advantageous as it allows good modules to stay in the module list, but it is also disadvantageous as it could possibly allow the evolution of the genotype to progress at a slower rate.

Table 1: The three nodes types and how the operators effect each of them

Node Type	Action of Compress	Action of Expand	Action of Genotype Point Mutation
0	Compress into module	Immune	Changes function or inputs
I	Immune	Expand into nodes	Changes inputs
II	Immune	Immune	Changes function or inputs

The module genotypes contained in the module list can also be evolved through the action of five different operators: *module point mutation*, *add-input*, *add-output*, *remove-input* and *remove-output*. The module point mutation operator is a restricted version of the ECGP genotype point mutation operator, as it can still mutate the inputs and function of any node encoded in the module genotype, but it is not allowed to introduce any type II nodes into the module genotype. It can also mutate which node output each of the module outputs are connected to (similar to program outputs in CGP). The add-input and add-output operators allow greater connectivity to and from the contents of a module, by increasing the number of module inputs or module outputs by one respectively each time either operator is applied, making a more generalized module. When the add-input operator is applied to a module, the gene representing the number of module inputs in the module header is incremented by one, and an extra gene is inserted into all nodes (type I and type II) representing the module in the genotype, as a randomly chosen value for the new module input. Likewise, when the add-output operator is applied to a module, the gene representing the number of module outputs in the module header is incremented by one, and an extra gene is added to the module output section of the module genotype, as randomly chosen values for the node index and node output that the new module output is connected to.

Alternatively, the remove-input and remove-output operators reduce the connectivity to and from the contents of a module, by decreasing the number of module inputs or module outputs by one respectively each time either operator is applied, therefore making a more specialized module. When the remove-output operator is applied to a module, the gene representing the number of module inputs in the module header is decremented by one, and the gene corresponding to the module input randomly chosen, is removed from all nodes (type I and type II) representing the module in the genotype. Likewise, when the remove-output operator is applied to a module, the gene representing the number of module outputs in the module header is decremented by one, and the gene corresponding to the randomly chosen module output is removed from the module output section of the module genotype. All of the operators: add-input, add-output, remove-input, and remove-output must comply with the restrictions on the number of module inputs and module outputs at all times. Further information about all of the module operators (including figures explaining their operation) is available in our previous work [15, 17, 16].

4. EXPERIMENT DETAILS

The parameters used for CGP and ECGP on both the Lawnmower and H-IFF problems are shown in Table 2. The probability values chosen for the ECGP operators were taken from [16].

4.1 Lawnmower Problem

The Lawnmower Problem was first introduced by Koza in his second book [6] to test the effectiveness of Automatically Defined Functions by exploiting the modularity of the lawnmower problem. Since then it has been used as a benchmark problem by many other researchers in the testing of new GP techniques and representations [9].

The concept of the lawnmower problem is to guide a lawnmower around a grass lawn, which consists of n x m squares (where n and m are user defined parameters). The lawnmower moves around the lawn one square at a time, and cuts all the grass in each square it visits. The lawnmower is allowed to revisit a square of the lawn as many times as it likes, but the grass in a square can only be cut by the lawnmower once, therefore by revisiting squares of the lawn, the lawnmower is using an inefficient approach to mowing the lawn. However, the lawn is a "magic" lawn, when the lawnmower moves off a square on any side of the lawn, it reappears in the square on the opposite side of the lawn. The lawnmower always starts in the centre square of the lawn and starts off by facing in a northward direction. The lawn is cut when every square has been visited by the lawnmower.

The movement of the lawnmower is controlled by a CGP or ECGP program. The program has three inputs which it can use: *move*, which moves the lawnmower one square forward on the lawn in the direction the lawnmower is facing, and cuts all of the grass in that square, *turn*, which rotates the lawnmower 90° clockwise in the current square on the lawn and *random constant*, which stores a randomly distributed vector for the entire run of the form [x, y], where $0 <= x < n$ and $0 <= y < m$. In conjunction with the operations just described, the move and turn inputs also return the vectors [0, 0], so that mathematical operations can take place on any combination of the inputs. For further details see [6].

The function set for the program consists of: *v8a*, which takes two vectors and returns the result from the addition of these two vectors, *frog*, which takes a vector [x, y] and jumps the lawnmower to another square on the lawn, a distance of x squares in the horizontal direction and y squares in the vertical direction away, and returns the vector [x, y], and *progn*, which takes two inputs and executes everything from the first input, and then everything from the second input, before returning the resulting vector from the second input.

The fitness function for this problem is defined as the number of squares on the lawn which are left uncut by the lawnmower, after the evolved program has been run once.

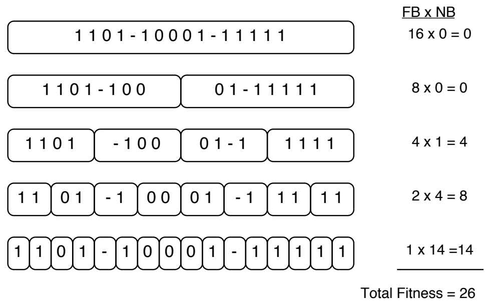

Figure 5: The H-IFF fitness function for a 16-bit string containing 0's, 1's and blanks(-). *FB* represents the fitness bonus at each level of the hierarchy for a correct block, and *NB* represents the number of correct blocks in the current level of the hierarchy.

For this problem we are minimizing the fitness value, as a lawn in which all the squares are cut would have a fitness score of zero, and would be a solution to the problem.

4.2 Hierarchical If-and-Only-If (H-IFF)

The Hierarchical If-and-Only-If Problem was proposed by Watson, Hornby and Pollack in the late nineties, as a more suitable problem for testing the performance of Genetic Algorithms using crossover. We suggest that this problem might be considered as a benchmark for GP techniques. In our case, we are using it to allow a comparison of CGP and ECGP and their scalability. H-IFF is based on the Building Block Hypothesis [4, 2] and groups bits into blocks, then these blocks into larger blocks, and continues until a hierarchy of blocks is formed. As you ascend the hierarchy, at each level the number of blocks halve whilst the size of the blocks double, until you reach the top block of the hierarchy which is the solution to the problem. There have been many other problems (for example the Royal Road functions [12, 8, 5]), which are constructed using the concepts of building blocks and have been used for testing Genetic Algorithms, but H-IFF differs from others by modelling the building block interdependency in a consistent hierarchical fashion. However, H-IFF has two possible solutions, a bit string containing all 0's and a bit string containing all 1's (therefore all building blocks also have two solutions). This leads to a flatter fitness landscape with multiple local optima and two global optima, which makes the problem much more difficult. Watson has suggested the H-IFF problem is very difficult to solve unless crossover is used [10]. Figure 5 describes the H-IFF fitness function in more detail, and the associated fitness rewards with each level of the hierarchy.

One of the main issues we faced was deciding how to apply CGP and ECGP to a problem which is designed for Genetic Algorithms. A method was needed which would scale well for different size bit-strings. The method eventually chosen was heavily influenced by the Lawnmower Problem, which was detailed in section 4.1. Instead of controlling the actions of a lawnmower on a two dimensional lawn, the CGP program controls a tape head on a piece of one dimensional tape, which is divided into n squares, where n is the length of the bit-string. The initial value of all the squares on the tape is blank, and the tape head starts in the centre of the

Table 2: The parameter settings used for CGP and ECGP in all of the experiments (* indicates ECGP only). The mutation rate is expressed as a percentage of the genotype length. All of the operator rates and probabilities are per generation.

Parameter	Value
Population size	5
Initial genotype size	200 nodes (600 genes)
Genotype point mutation rate	3% (18 Genes)
Genotype point mutation probability	1
Compress/Expand probability *	0.1/0.2
Module point mutation probability *	0.04
Add/Remove input probability *	0.01/0.02
Add/Remove output probability *	0.01/0.02
Maximum module size *	5 or 10 nodes
Module list initial state *	Empty

tape. Similar to the lawnmower problem, the tape in the H-IFF problem is a "magic" tape, when the tape head moves off one end of the tape it reappears in the square at the opposite end of the tape. When the tape head visits a square, it changes the squares value according to the rule:

$$\begin{aligned} &if(x == blank),\ x = 0 \\ &if(x == 0),\ x = 1 \\ &if(x == 1),\ x = 0 \end{aligned} \tag{1}$$

where x is the value of the square. This operation is the same as a bit flip operator once a square contains a value. When the tape head has finished, it will have produced a bit-string of length n containing the symbols: - (blank), 0 and 1, which can then be evaluated using the H-IFF function and assigns a fitness value to the CGP program.

The CGP program which controls the tape head takes three program inputs: *move*, which moves the tape head one square in the direction it is facing, and changes the value of the new square according to Equation 1, *turn*, which alters the direction in which the tape head travels along the tape from right to left or vice versa, and *random constant*, which stores the value of a random number, r, chosen at the start of each independent run, where $0 <= r < n$. Both move and turn also return the constant 0 so that mathematical operations can be performed on the program inputs.

The function set used by CGP is also reminiscent of the lawnmower problem as it uses the same functions: *progn*, a program node which executes the graph connected to its first input, followed by the graph connected to its second input and returns the result of the second input, *v8a*, which performs addition on the values of its two inputs and returns the result and *frog*, which jumps the tape head to a new square on the tape, a number of squares specified by its input in the direction the tape head is facing, and alters the value in the new square according the rule in Equation 1.

5. RESULTS

For all experiments, the computational effort was calculated using the formula in Equation 2 from [3] with z=99%.

$$\begin{aligned} P(M,i) &= \frac{N_s(i)}{N_{total}}, \\ R(z) &= ceil\left(\frac{\log(1-z)}{\log(1-P(M,i))}\right), \\ I(M,i,z) &= MR(z)\,i+1 \end{aligned} \tag{2}$$

The computational effort figures for CGP and ECGP applied to the lawnmower and H-IFF problems are shown in Tables 4 and 3 respectively. In Table 4, the computational effort figures for Parallel Distributed GP (PDGP) and GP (with and without Automatically Defined Functions (ADFs)) were taken from [9] and [6] respectively. Statistics for the average number of fitness evaluations and the standard deviation are also included in Table 3.

For both the lawnmower and H-IFF problems, all fifty independent runs of CGP and ECGP produced 100% successful solutions.

For all lawn sizes of the lawnmower problem, it can be seen that the performance of CGP and ECGP starts off fairly evenly for the smaller lawn sizes but as the lawn size increases, ECGP starts to perform better than CGP. In ECGP, the speedup grows with problem difficulty, suggesting that ECGP could perform even better on larger problems. This speedup can be attributed to the discovery and re-use of sub-routines, which allow the lawnmower to cut multiple numbers of grass squares covering an area of the lawn and then allowing the same pattern to be repeated elsewhere on the lawn. This supports the previous findings of ECGP in [15, 17, 16].

Comparing the computational effort figures for CGP with Parallel Distributed GP (up to a lawn size of 128) and GP without Automatically Defined Functions (up to a lawn size of 96), it can clearly be seen that CGP performs better than both techniques. CGP performs between 2.2 and 3.1 times faster than Parallel Distributed GP and between 14.8 and 1831.4 times faster than GP without Automatically Defined Functions. In fact, CGP even outperforms GP with Automatically Defined Functions on this problem. This result emphasises the performance gain of using a graph based representation (as in CGP and Parallel Distributed GP), rather than a tree based representation (as in GP). It can also be seen from comparing the two techniques which are capable of reusing sub-routines, ECGP and GP with Automatically Defined Functions, that ECGP performs between 3.9 and 12.5 times faster than GP with Automatically Defined Functions. Notice also that the speedup grows with the size of the lawn, indicating that ECGP may perform even better than GP with Automatically Defined Functions on larger problems.

For all lengths of bit-string in the H-IFF problem, CGP performs better than ECGP. CGP also seems to scale better than ECGP as the length of the bit-string increases,

Table 3: The computational effort, average number of fitness evaluations (square brackets) and standard deviation (round brackets) figures for CGP and ECGP (with a maximum module size of 3, 5 and 8) applied to the H-IFF problem. All figures are in terms of thousands.

Length	CGP	ECGP-3	ECGP-5	ECGP-8
8	1.0 [0.2] (0.3)	1.1 [0.3] (0.4)	1.0 [0.2] (0.3)	1.1 [0.4] (0.8)
16	2.7 [4.0] (12.6)	1.9 [0.7] (0.8)	3.4 [4.8] (16.8)	2.9 [1.3] (2.4)
32	3.8 [76.3] (516.6)	4.3 [15.5] (70.3)	4.2 [138.4] (944.3)	4.2 [7.9] (25.8)
64	5.6 [4.7] (13.6)	6.7 [231.9] (1,612.5)	7.7 [57.6] (293.1)	7.7 [16.7] (91.3)
128	5.8 [2.4] (5.2)	7.0 [103.5] (719.6)	9.0 [5.0] (18.0)	8.8 [43.7] (292.4)
256	8.3 [2.3] (2.5)	11.5 [6.0] (14.9)	19.2 [25.0] (130.4)	15.8 [11.9] (34.7)

Table 4: The computational effort (in terms of thousands) and speedup figures for CGP, ECGP, PDGP and GP for the Lawnmower problem.

Lawn Size	CGP (1)	ECGP (2)	PDGP (3)	GP(No ADFs) (4)	GP(With ADFs) (5)	Speedup (1)&(2)	Speedup (1)&(3)	Speedup (1)&(4)	Speedup (2)&(5)
32	1.3	1.3	4	19	5	1.0	3.1	14.8	3.9
48	1.6	1.6	5	56	9	1.0	3.1	35.0	5.6
64	2.4	1.6	5	100	11	1.5	2.1	41.6	6.9
80	1.9	1.9	5	561	17	1.0	2.6	291.9	8.8
96	2.6	1.6	6	4,692	20	1.6	2.3	1831.4	12.5
112	2.6	1.9	6	-	-	1.3	2.3	-	-
128	3.2	2.2	7	-	-	1.4	2.2	-	-
144	2.6	1.9	-	-	-	1.3	-	-	-
160	3.2	1.9	-	-	-	1.7	-	-	-
176	2.9	1.9	-	-	-	1.5	-	-	-
192	3.5	2.6	-	-	-	1.4	-	-	-
208	2.9	2.2	-	-	-	1.3	-	-	-
224	3.8	2.9	-	-	-	1.3	-	-	-
240	4.2	2.6	-	-	-	1.6	-	-	-
256	3.5	1.9	-	-	-	1.8	-	-	-

suggesting that CGP may perform better than ECGP on even longer bit-strings. The performance difference between ECGP and CGP could be attributed to the overhead of module acquisition, evolution and re-use in ECGP not being able to find and exploit any modularity in the program, which generates the bit-string. Alternately, the complexity of the problem could be too low, suggesting ECGP requires more time to discover and learn how to use good modules than CGP does to find a solution.

The results in Table 3 only compare CGP and ECGP, as no other GP technique has been applied to this problem. The only published work with any results for the H-IFF problem is by Watson et al [10], which states the results of a Genetic Algorithm (GA) with 2-point crossover on the 64-bit H-IFF problem. The GA with 2-point crossover applied to the 64-bit H-IFF problem, only reached a fitness of 358 out of a possible 448 when it had reached 1000 generations. It also used a population size of 1000 with elitism of 1%. The results were averaged over ten runs. Using these figures it is possible to calculate a rough computational effort figure for the result using Equation 2. If we are really generous and say that by generation 400 (as the fitness did not change after this), the GA had actually solved the problem, and give the computational effort calcualtion the best possible *R(z)* value of 1, then we can calculate the computational effort of the GA as shown in Equation 3, where the value of 990 is established from 1000 - 1% (10 generations).

$$CE = 1 * 990 * 400 = 396,000 \tag{3}$$

Comparing the computational effort figures for the GA with those of CGP or ECGP for the 64-bit H-IFF problem, it can clearly be seen that CGP and ECGP perform significantly better (by a factor of approximately 71 or 59 respectively) than the GA on the 64-bit H-IFF problem. Comparing the average number of fitness evaluation statistics of all three techniques also shows a similar trend. The results of CGP and ECGP are contrary to the views expressed by Watson et al [10]. Watson et al believe that crossover is required to solve the H-IFF problem. Neither CGP or ECGP use any form of crossover operator, they are both mutation based. The implicit re-use of nodes in the graph-based representation of CGP and ECGP means a mutation in the genotype can cause changes of varying magnitude in the phenotype. We think that our favourable results are related to the beneficial properties of the genotype-phenotype mapping used in CGP and ECGP, particularly the use of neutrality.

6. CONCLUSION

In this paper, we have presented for the first time the application of CGP and ECGP to two problems (the lawnmower and H-IFF problems) over a range of problem sizes. On the lawnmower problem, both CGP and ECGP have been shown to perform better than GP (with and without Automatically Defined Functions) and Parallel Distributed GP. ECGP is also shown to perform better than CGP and the speedup grows with problem difficulty, indicating ECGP may perform even better than CGP on larger, more difficult problems (a similar trend was also found when comparing ECGP with GP with Automatically Defined Functions). This follows a very similar trend to the results found in the previous work on ECGP [15, 17, 16].

On the H-IFF problem, CGP surprisingly outperforms ECGP (with varying maximum module sizes) over all lengths of bit-string tested. However, both CGP and ECGP significantly outperform the GA with 2-point crossover from [10], based on our approximate computational effort figures.

It was also found that the maximum module size chosen for ECGP can drastically affect performance and will be investigated further in future investigations. Currently ECGP does not allow modules within modules. However, we do have a working version of ECGP that allows embedded sub-modules but we are currently investigating the problem of bloat within the embedded sub-modules found in the inactive areas of the module genotype. When a solution is found, we intend to allow embedded sub-modules in future work, as this could lead to an even greater boost in performance.

7. REFERENCES

[1] P. J. Angeline and J. Pollack. Evolutionary Module Acquisition. In *Proc. of the 2nd Annual Conference on Evolutionary Programming*, pages 154–163, La Jolla, CA, USA, 25-26 Feb. 1993.

[2] D. E. Goldberg. *Genetic Algorithms in Search, Optimisation and Machine Learning*. Addison-Wesley, Reading, MA, USA, 1989.

[3] A. Dessi, A. Giani, and A. Starita. An analysis of automatic subroutine discovery in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 996–1001, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[4] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, USA, 1975.

[5] J.H. Holland. Royal Road Functions. *Internet Genetic Algorithms Digest*, 7(22), 1993.

[6] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, USA, 1994.

[7] J. F. Miller and P. Thomson. Cartesian Genetic Programming. In *Proc. of the 3rd European Conference on Genetic Programming*, volume 1802 of *LNCS*, pages 121–132, Edinburgh, 15-16 Apr. 2000. Springer-Verlag.

[8] M.Mitchell and S. Forrest and J.H. Holland. The Royal Road for Genetic Algorithms: Fitness Landscapes and GA Performance. In *Proc. of 1st European Conference on Artifical Life*, Cambridge, MA, USA, 1992. MIT Press.

[9] R. Poli. Parallel Distributed Genetic Programming. Technical Report CSRP-96-15, School of Computer Science, University of Birmingham, B15 2TT, UK, Sept. 1996.

[10] R. A. Watson and G. S. Hornby and J. B. Pollack. Modelling Building Block Interdepenedancy. In *Parallel Problem Solving from Nature: 5th International Conference*, volume 1498 of *LNCS*, pages 97–108, Amsetrdam, The Netherlands, sep 1998. Springer-Verlag.

[11] J. Rosca. Towards automatic discovery of building blocks in genetic programming. In *Working Notes for the AAAI Symposium on Genetic Programming*, pages 78–85, MIT, Cambridge, MA, USA, 10–12 Nov. 1995. AAAI.

[12] S. Forrest and M. Mitchell. Relative Building-block Fitness and the Building-block Hypothesis. In *Foundations of Genetic Algorithms 2*. Morgan Kaufmann, San Mateo, CA, USA, 1993.

[13] L. Spector. Simultaneous evolution of programs and their control structures. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 7, pages 137–154. MIT Press, Cambridge, MA, USA, 1996.

[14] L. Spector. Autoconstructive evolution: Push, pushGP, and pushpop. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 137–146, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[15] J. A. Walker and J. F. Miller. Evolution and Acquisition of Modules in Cartesian Genetic Programming. In *Proc. of the 7th European Conference on Genetic Programming*, volume 3003 of *LNCS*, pages 187–197, Coimbra, Portugal, 5-7 Apr. 2004. Springer-Verlag.

[16] J. A. Walker and J. F. Miller. Improving the Evolvability of Digital Multipliers using Embedded Cartesian Genetic Programming and Product Reduction. In *Proc. of the 2005 International Conference on Evolvable Systems*, volume 3637 of *LNCS*, pages 131–142, Sitges, Spain, 12-14 Sept. 2005. Springer-Verlag.

[17] J. A. Walker and J. F. Miller. Investigating the Performance of Module Acquisition in Cartesian Genetic Programming. In *Proc. of the 2005 Genetic and Evolutionary Computation Conference*, volume 2, pages 1649–1656, Washington DC, USA, 25-29 June 2005. ACM Press.