Investigating the Performance of Module Acquisition in Cartesian Genetic Programming

James Alfred Walker
University of York
Heslington, York,
YO10 5JA, UK.
Jaw500@ohm.york.ac.uk

Julian Francis Miller
University of York
Heslington, York,
YO10 5JA, UK.
jfm@ohm.york.ac.uk

ABSTRACT
Embedded Cartesian Genetic Programming (ECGP) is a form of the graph based Cartesian Genetic Programming (CGP) in which modules are automatically acquired and evolved. In this paper we compare the efficiencies of the ECGP and CGP techniques on three classes of problem: digital adders, digital multipliers and digital comparators. We show that in most cases ECGP shows a substantial improvement in performance over CGP and that the computational speedup is more pronounced on larger problems.

Categories and Subject Descriptors
I.2.2 [Artificial Intelligence]: Automatic Programming – Program Synthesis.

General Terms
Algorithms, Design, Performance.

Keywords
Cartesian Genetic Programming, Module Acquisition, Modularity, Digital Adders, Digital Comparators, Digital Multipliers, Computational Effort.

1. INTRODUCTION
Since the invention of Genetic Programming (GP) by John Koza [3, 4], researchers in the GP community have been constantly looking for new ways to make it a more powerful tool, so GP can be applied to larger and more complex problems. In the late nineties, one such technique called Cartesian Genetic Programming (CGP) was proposed [7], which used a directed graph representation instead of the standard tree based representation of GP. This alternative representation gave the technique a number of advantages over the traditional GP approach. Even though CGP did not have the equivalent of automatically defined functions it was empirically demonstrated to be more computationally efficient than GP with Automatically Defined Function's (ADFs) [6] on a number of problems. Since then CGP has been developed further by incorporating elements taken from a technique known as evolutionary module acquisition that allows the dynamic acquisition, evolution and re-use of modules [15]. We call the new method Embedded Cartesian Genetic Programming (ECGP). Its main feature is that it uses CGP to construct modules that can be called from the main CGP code. Although at present this technique has not allowed modules to have sub modules embedded ECGP has been shown to perform better than standard CGP on a series of parity problems (even 4 to even 8 parity) [15]. In this paper we are trying to substantiate the results found so far using ECGP by applying the technique to three more problem classes: digital adders, digital multipliers and digital comparators. We also present computational effort figures for both techniques on these problems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'05, June 25-29, 2005, Washington, DC, USA.
Copyright 2005 ACM 1-59593-010-8/05/0006…$5.00.

As with even-parity functions evolving digital adders using GP with the primitive Boolean operations of AND, OR, NAND and NOR is very difficult. In addition, like the even parity problem, digital adders are particularly appropriate for testing module acquisition techniques as the problem can be more compactly represented when the XOR Boolean function is available. It is well known that all digital adders can also be constructed using a series of 1-bit adders, a technique known as Ripple Carry Addition.

The evolution of digital multipliers is another even more difficult problem to solve as the complexity of the problem scales rapidly with the number of inputs [14]. It is also an interesting problem to solve using a modular technique, as no one knows if a modular general solution exists for all digital parallel multipliers, other than a cellular array of adders. We also introduce the digital comparator problem to the GP community as a new potential benchmark.

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 are shown in section 4 followed by the results and comparisons for all three experiments in section 5. Section 6 gives conclusions and some suggestions for future work.

2. RELATED WORK ON MODULE ACQUISITION, AUTOMATICALLY DEFINED FUNCTIONS AND MACROS
The original idea of Module Acquisition [1] was to try and find a way of protecting desirable partial solutions contained in the genotype, in the hope that it might be beneficial in finding a solution. This is because in practice you may find a desirable partial solution in the genotype, but due to the nature of evolution,

an operator could modify the partial solution therefore causing the program to take longer to find a solution.

Module acquisition does this by introducing another 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. However they affect the possible offspring that might be generated using evolutionary operators. Atomisation [1] not only makes sections of the genotype immune from manipulation by operators but also represents the module as a new component in the genotype therefore allowing the module to be manipulated further by additional compress operators. This allows the possibility of having modules within modules therefore creating a hierarchy organisation of modules. These techniques have been shown to decrease the time taken to find a solution by reducing the amount of manipulations that can take place in the genotype. Rosca's method of Adaptive Representation through Learning (ARL) [9] also extracted program segments that were encapsulated and used to augment the GP function set. The system employed heuristics that tried to measure good program code and also methods to detect when search had reached local optima from population fitness statistics. In the latter case the extracted functions could be modified. More recently Dessi et al [2] showed that *random* selection of program sub-code for re-use is more effective than other heuristics across a range of problems. Also they concluded that, in practice, ARL does not produce highly modular solutions. Once the contents of modules are themselves allowed to evolve (as in ECGP) they become a form of Automatically Defined Function (ADF), however in contradistinction to Koza's form of ADFs [4] and Spector's Automatically Defined Macros [10], there is no explicit specification of the number or internal structure of such modules. This freedom also exists in Spector's more recent PushGP [11].

In addition to decreasing computational effort and making more modular code van Belle and Ackley have shown that ADFs can increase the evolvability of populations of programs over time [12]. They investigated the role of ADFs in evolving programs with a time dependent fitness function and found that not only do populations recover more quickly from periodic changes in the fitness function, but the recovery rate increases in time as the solutions become more modular. Woodward [16] showed that the size of a solution is independent of the primitive function set used when modularity is permitted, thus including modules can remove any bias caused by the chosen primitive function set.

3. EMBEDDED CARTESIAN GENETIC PROGRAMMING (ECGP)

3.1 Representation

Embedded Cartesian Genetic Programming [15] is an extension of Cartesian Genetic Programming [6][7], which was originally developed for the automatic evolution of digital circuits [5]. The technique is also similar to Parallel Distributed GP, independently developed by Poli [8]. Both CGP and ECGP share the same structure and represent a program as a directed graph (that for feed-forward functions is acyclic). 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 genes) in which the number of nodes in the program (phenotype) can vary but is bounded. In ECGP the *genotype* is a variable length representation (in terms of genes and nodes) in which the number of nodes and genes in the graph is bounded. The variable number of nodes in the ECGP genotype is the result of the compression and expansion of modules and the variable number of genes is a result of the re-use of modules and the module mutation operators. In Figure 1 an example of the differences between a CGP and an ECGP genotype are shown. Despite these differences, both CGP and ECGP are initialized with a CGP style genotype (i.e. no modules). However, we emphasize, both techniques use a genotype-phenotype mapping that does not require all nodes to be connected to each other. This results in a bounded variable length phenotype.

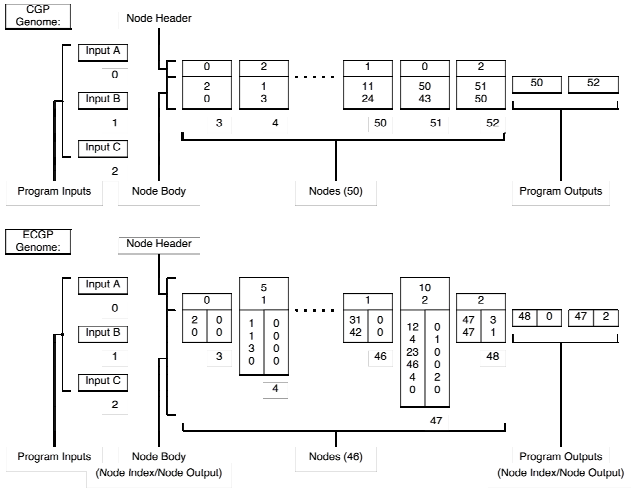

Figure 1. Examples of evolved CGP and ECGP genotypes for the 1-bit adder problem (3 inputs, 2 outputs). Both genotypes were initialized with 50 nodes (150 genes). The top or only number in the node headers represents the function; the remaining number (where present) is the node type. The node body represents the node inputs, which in ECGP are split into two parts: the node index in the genotype and the points from which the node outputs are taken from. The node index is shown underneath each node.

Each of the nodes consists of two parts: a node header and a node body. The node header encodes the function (primitive or module) of the node and the type of the node (type I or type II) if the node represents a module (the concept of module type is explained in section 3.4). The node body encodes the inputs of the node. Each input is encoded by two integers: one represents the *index* of the node or program input (terminal) in the genotype and the other represents the *output* of the node (note nodes can have multiple outputs). The number of inputs and outputs that each node has is dictated by the arity of its function. To clarify still further, consider the genotype fragment of ECGP from Figure 1 which we have extracted (Figure 2):

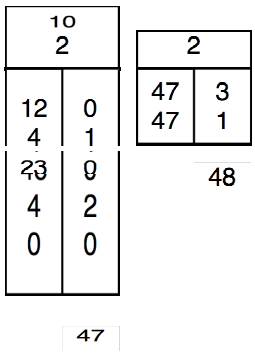

Figure 2. A fragment of the ECGP genotype shown in Figure 1. Node 47 with function 10 is of type II (see later) and is a module with 6 inputs connected to node indexes 12(0), 4(1), 23(0), 46(0), 4(2), 0(0). The numbers in brackets are the particular outputs of the nodes. Node 48 with function 2 is a primitive function whose inputs are both taken from node 47. The first input comes from output 3 and the second comes from output 1.

The nodes take their inputs in a feed forward manner from either the output of a previous node or from one of the program inputs (terminals). The program inputs are numbered from *0* to *n-1* where *n* is the number of program inputs. The nodes in the genotype are then also numbered sequentially starting from *n* to *n+m-1* where *m* is the user-determined upper bound of the number of nodes. These numbers are used for referencing the outputs of the nodes and the program inputs. If the problem requires *k* program outputs then *k* integers are added to the end of the genotype, each one representing 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 and can be altered by point mutation. In Figure 3 an ECGP genotype is shown and how it is decoded (a 1-bit digital adder circuit).

Although each node must have a function and a set of inputs for that function, the outputs of a node do not have to be connected. This is shown in Figure 3, where the output of nodes 6 and 8 are not used (shown in grey dashed lines). This causes areas of the genotype to remain dormant, leading to a neutral effect on genotype fitness (neutrality). When point mutations are carried out on genes representing connections (the mutation is constrained to respect the directed and acyclic nature of the graphs) these dormant genes can be activated or active genes can be made dormant. This unique type of neutrality has been investigated in detail [7, 13, 17] and in the problems studied, found to be extremely beneficial to the evolutionary algorithm.

3.2 Evolutionary Strategy

The evolutionary algorithm used for the experiments is a form of $1+\lambda$ evolutionary strategy, where $\lambda=4$, i.e. one parent with 4 offspring (population size 5). The algorithm is as follows:

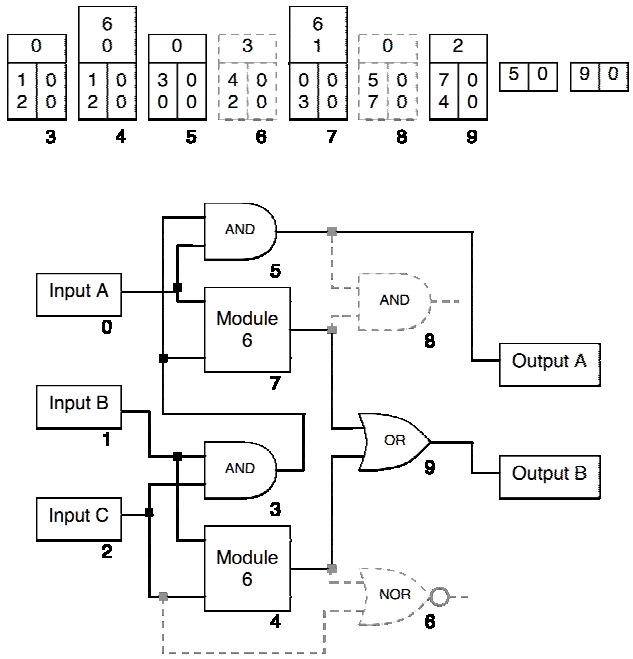

Figure3. An ECGP genotype and the corresponding phenotype for a 1-bit digital adder circuit. The module occurring twice in the genotype represents a possible structure for an XOR Boolean function constructed from the function set. The dormant areas of the genotype and phenotype are shown in grey dashes. Note also that node functions 1, 4 and 5 are not used in this example.

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:
   - If any offspring has a better fitness; the best becomes the winner.
   - Otherwise, an offspring with the same fitness as the best is randomly selected.
   - 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.

3.3 Module Representation

A module is represented as a bounded variable length genotype that has the same characteristics of a standard CGP genotype and has the same genotype-phenotype mapping, resulting in a bounded variable length phenotype. The genotype consists of a list of integers and is split into two parts: the module header and the module body. The first part contains four integers and represents the module header which stores information about the module such as the module identifier number, the number of module inputs, the number of nodes contained in the module and the number of module outputs. The second part of the module genotype is the module body, which 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 standard CGP genotype. An example of a module genotype showing the separate components is shown in Figure 4.

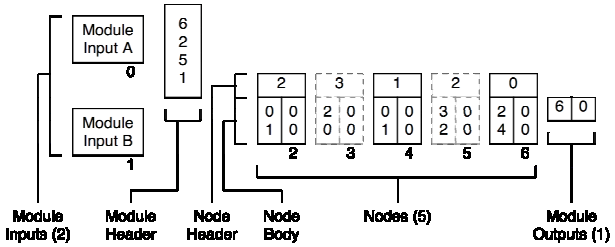

Figure 4. An example of a module genotype. The four numbers in the module header represent the module identifier number, the number of inputs, the number of nodes and the number of outputs of the module respectively. The nodes are represented the same as in ECGP. The module output represents which node the module takes its output from.

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 nodes (any fewer and it would either be an empty module or a primitive function) and a predefined 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 access the module) and a maximum of n module outputs, where n 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 $2n$ module inputs, where n is equal to the number of nodes contained in the module. 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. We aim to remove this restriction in future work.

An Example of the genotype and corresponding phenotype of a module can be seen in Figure 5.

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. An example of the module list can be seen in Figure 6. The module list is dynamic and has no restrictions on its maximum size and is updated every generation when the fittest individual 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 module list so that bloat never occurs.

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 (including a module point operator – see section 3.4).

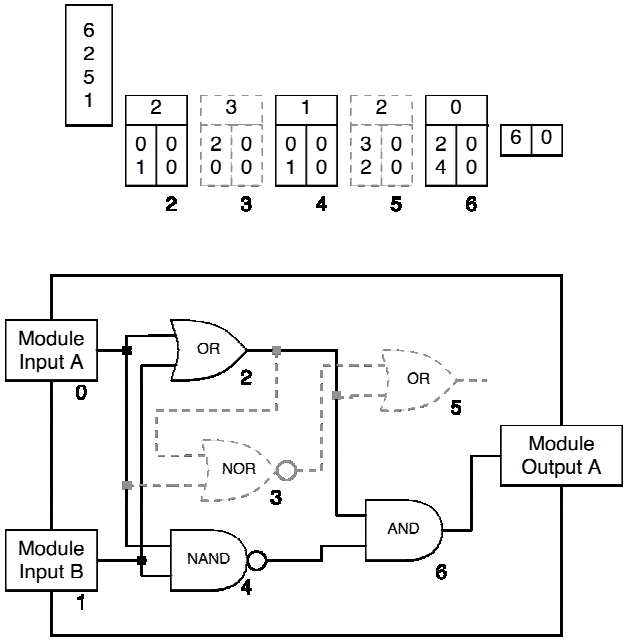

Figure 5. The genotype and corresponding phenotype of a module representing an XOR Boolean function. The dormant areas of the genotype and phenotype are shown in grey dashes.

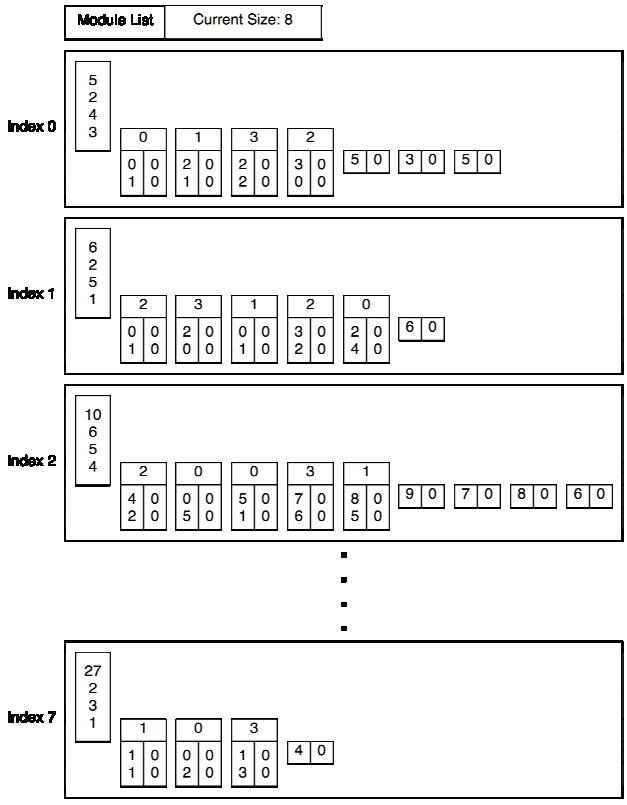

Figure 6. An example of a possible module list currently containing eight module genotypes.

3.4 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 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 between these two points into a new module, which is encoded into a module genotype as described earlier. Note, that if there any modules 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 that 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 of an individual as a type I node. Any type I node is immune from the point mutation operator and remains in the genotype of an individual until it is removed by the expand operator (see Table 1).

The *expand* operator destroys a type I node by replacing it in the genotype of an individual with the nodes contained in the module that the type I node represented. The inputs of all of the latter nodes in the genotype of the individual are updated after the compress or expand operator has been applied so that all the connections remain intact. The reason for this is that the compress and expand operators only make a structural change to the genotype of an individual and have no affect on genotype fitness, as the genotypes 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 the genotype of an individual through the action of the *point mutation* operator. This is identical to that used in CGP with the exception that it can mutate the function of a node 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 and is treated like a standard node. This means the point mutation operator can also mutate the function of a type II node to any of the pre-defined functions or any available modules in the module list. It can also mutate any of the inputs of the type II node in the same way it would mutate the inputs of a standard node. If the function of a standard node or type II node is mutated, the new node keeps however many of the original node's inputs it needs and randomly generates any extra inputs it may require. 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 I and II are as follows:

Table 1. Node types and their properties

Node Type	Action of Compress	Action of Expand	Action of Genotype Point Mutation
I	Creation	Destruction	Changes node inputs
II	Immune	Immune	Creation or destruction or changes node inputs

The *module genotypes* contained in the module list can also be evolved through the action of five different operators: *point mutation*, *add-input*, *add-output*, *remove-input* and *remove-output*. The point mutation operator is the same as the CGP point mutation operator, as it can mutate the inputs and function of any node contained 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.

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 part of the module genotype is incremented by one and an extra gene is inserted into all nodes (type I and type II) representing the module in the genotype of the individual, 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 part of the module genotype is incremented by one and two extra genes are 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 part of the module genotype 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 of an individual. Likewise, when the remove-output operator is applied to a module, the gene representing the number of module outputs in the module header part of the module genotype is decremented by one and the two genes corresponding to the randomly chosen module output are 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.

4. EXPERIMENT DETAILS

The performance of CGP and ECGP was tested on three different classes of problem: digital adder, digital multiplier and digital

comparator. The *n*-bit digital adder problem takes two *n*-bit numbers and a 1-bit carry-in ($2n + 1$ program inputs) and adds them together to produce an *n*-bit number and a 1-bit carry-out ($n + 1$ program outputs). In this experiment we used the 1-bit, 2-bit and 3-bit digital adders. The *n*-bit digital multiplier problem is of a similar nature to the digital adder problem as it takes two *n*-bit numbers ($2n$ program inputs) and multiplies them together to produce a $2n$-bit number ($2n$ program outputs). Here we tested only the 2-bit and 3-bit digital multipliers (the 1-bit digital multiplier is just the AND Boolean function). The *n*-bit digital comparator problem is a new problem to the GP community but not unfamiliar to programmers. It takes two *n*-bit numbers ($2n$ program inputs) and compares them to produce a value of "1" at one of its three outputs and "0" at the other two outputs, depending on whether the first *n*-bit number is less than, equal to or greater than the second *n*-bit number. For this experiment we looked at the 1-bit, 2-bit and 3-bit digital comparators.

Table 2. The parameters settings used for CGP and ECGP in all of the experiments. The operator rate is expressed as a percentage of the genotype length. Both the operator rates and probabilities are per generation.

Parameter	Value
Population size	5
Initial genotype size	100 nodes (300 genes)
Function set for digital adder and digital comparator	{AND, NAND, OR, NOR}
Digital multiplier function set	{AND, AND with one input inverted, XOR}
Genotype point mutation rate	2% (6 genes)
Genotype point mutation probability	1
Compress probability	0.1
Expand probability	0.2
Module point mutation probability	0.04
Add-input probability	0.01
Add-output probability	0.01
Remove-input probability	0.02
Remove-output probability	0.02
Maximum module size (ECGP)	5 nodes
Initial module list contents	Empty
Number of independent runs	50

The fitness is defined as the number of phenotype output bits that differ from the perfect *n*-bit digital adder, *n*-bit digital multiplier or *n*-bit digital comparator function. A perfect solution has score zero.

The parameter settings used for CGP and ECGP in all of the experiments are shown in Table 2. The probability values chosen for the ECGP operators were found to be optimal by a trial and error process in previous ECGP experiments. The results for all CGP and ECGP experiments were averaged over fifty independent runs.

5. RESULTS

For all experiments, the Computational Effort (CE) was calculated using the formula found in [3] with z=99%. The number of hits was recorded every 40 generations to calculate the probability. The CE figures for the results of CGP and ECGP applied to all of the classes of problem chosen is shown in Table 3 below. These CE figures are only relevant when comparing CGP and ECGP with the same number of nodes in their genotypes and the same rate for the default point mutation. This is because CE figures for CGP and ECGP vary significantly depending on these values, therefore potentially causing an unfair comparison. We have only compared the CE figures of ECGP with CGP because no other researchers have provided CE figures for their GP techniques on these three classes of problem.

Table 3. The computational effort figures for CGP and ECGP for the digital adder, digital multiplier and digital comparator problems.

	CGP	ECGP	Speedup
1-Bit Adder	26,720	35,840	0.75
2-Bit Adder	493,760	203,520	2.43
3-Bit Adder	2,599,360	1,530,880	1.70
2-Bit Multiplier	35,840	35,520	1.01
3-Bit Multiplier	8,659,840	1,917,760	4.52
1-Bit Comparator	2,880	3,200	0.90
2-Bit Comparator	78,880	87,360	0.90
3-Bit Comparator	466,880	520,320	0.90

Over all eight problems tested, both CGP and ECGP produced 100% successful solutions over all runs. For the digital adder problem, the results show that ECGP performs between 1.48 and 2.43 times faster than CGP for the larger, more complex digital adder problems but on the simpler 1-bit digital adder CGP performs better, which could be attributed to the overhead of dynamic acquisition, evolution and re-use of modules in ECGP. The speedup by ECGP on the 2-bit and 3-bit digital adder problems can be accounted to the modules finding and re-using functions that are not defined in the primitive function set and make the problem a lot easier to solve. Two such functions are the XOR Boolean function and the half adder circuit, which consists of the XOR and AND Boolean functions. Both of these can be seen as partial solutions.

The results for the digital multiplier show that ECGP performs between 1.01 (which is negligible) and 4.52 times faster than CGP. The speedup factor also increases with problem size, indicating that ECGP may perform substantially better on even larger problems. This difference in performance between ECGP and CGP may also indicate that there exists some form of modularity in the digital multiplier problem that the modular nature of ECGP exploits. We do not believe that any

undiscovered modular way of building multipliers from multipliers was found but that ECGP simply found and re-used functions that can be constructed from the primitive function set and conform to the conventional way of building multipliers. Three such functions are the half adder (as found also in the digital adders), the 1-bit adder, which is constructed from two half adders and an OR Boolean function (as shown in Figure 2) and the 2-bit x 1-bit multiplier, which is simply constructed from two AND Boolean functions. All three of these functions are considered as partial solutions.

CGP outperforms ECGP on each of the digital comparator problems by a constant speedup value each time. This could have happened because any modularity that may exist in the comparator circuit is not being exploited, possibly due to the maximum module size being set too small. Another possibility is that there may not be any modularity in the comparator circuit and the difference in performance is also due to the dynamic acquisition, evolution and re-use of modules. If this is the case then it could show that using the added features of ECGP when they are not beneficial to solving the given problem only slows performance down by 10% when compared with CGP. Therefore it may be possible to say that if you apply ECGP to a problem that may or may not contain any form of modularity, ECGP could perform anywhere between 10% slower and a number of times faster than CGP. This is an interesting theorem and will be investigated further in future work.

All of the experiments were run on a single processor desktop PC with 512MB of memory. The time taken to complete 50 runs of each problem varied between a few minutes to a few hours depending on the difficulty of the problem. ECGP only took fractionally longer to complete one thousand generations on any problem than CGP showing that the computational time required for the overhead of module acquisition is quite small and the computational time taken for fitness evaluation (both CGP and ECGP) is by far the dominant factor.

6. CONCLUSION

We have presented for the first time the application of ECGP to three classes of difficult problems: digital adders, digital multipliers and digital comparators. ECGP shows a significant speedup when compared with non-modular CGP on the digital adder and digital multiplier problems but did not perform as well as CGP on the digital comparator problem by a constant amount each time. This phenomenon found in the comparator problem will be investigated further in future work to see if an explanation can be found to account for this behavior. The results from the digital multiplier problem also indicate that ECGP may perform substantially better than non-modular CGP on even larger problems.

Currently ECGP does not allow modules within modules. However, we do have a working ECGP prototype program that does allow embedded sub-modules but we are currently investigating the problem of bloat found within the unused areas of the module genotype that contain embedded sub-modules. When a solution is found to this problem, we intend to allow embedded sub-modules in future work as this could lead to an even greater boost in performance.

7. REFERENCES

[1] Angeline, P. J. Pollack, J. (1993) Evolutionary Module Acquisition, Proceedings of the 2nd Annual Conference on Evolutionary Programming, pp. 154-163, MIT Press, Cambridge.

[2] Dessi, A. Giani, A. Starita, A. (1999) An Analysis of Automatic Subroutine Discovery in Genetic Programming, GECCO 1999: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 996-1001, Morgan-Kaufmann, San Francisco.

[3] Koza, J. R. (1993) Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, London.

[4] Koza, J. R. (1994) Genetic Programming II: Automatic Discovery of Reusable Programs, MIT Press, London.

[5] Miller, J. F. , Thomson, P., and Fogarty T. C. (1997) Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study, Genetic Algorithms and Evolution Strategies in Engineering and Computer Science: Recent Advancements and Industrial Applications. Editors: D. Quagliarella, J. Periaux, C. Poloni and G. Winter, Wiley.

[6] Miller, J. F. (1999) An Empirical Study of the Efficiency of Learning Boolean Functions using a Cartesian Genetic Programming Approach, GECCO 1999: Proceedings of the Genetic and Evolutionary Computation Conference, Orlando, Florida, pp 1135-1142, Morgan Kaufmann, San Francisco.

[7] Miller, J. F. Thomson, P. (2000) Cartesian Genetic Programming, Proceedings of the 3rd European Conference on Genetic Programming, Edinburgh, Lecture Notes in Computer Science, Vol. 1802, pp 121-132, Springer-Verlag, Berlin.

[8] Poli, R. (1996), Parallel Distributed Genetic Programming, Technical Report, CSRP-96-15, University of Birmingham, UK.

[9] Rosca, J. P. (1995) Genetic Programming Exploratory Power and the Discovery of Functions, Proceedings of the 4th Annual Conference of Evolutionary Programming, San Diego, pp 719-736, MIT Press, Cambridge.

[10] Spector, L. (1996) Simultaneous evolution of programs and their control structures, Advances in Genetic Programming II, pp. 137-154, MIT Press, Cambridge.

[11] Spector, L. (2001) Autoconstructive Evolution: Push, PushGP, and Pushpop, Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001, pp. 137-146. San Francisco, CA: Morgan Kaufmann Publishers

[12] Van Belle, T, and Ackley, D.H. (2001) Code Factoring and the Evolution of Evolvability, Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001, pp. 1383--1390. San Francisco, CA: Morgan Kaufmann Publishers

[13] Vassilev, V. K. and Miller J. F. (2000) The Advantages of Landscape Neutrality in Digital Circuit Evolution, Proceedings of the 3rd International Conference on Evolvable Systems: From Biology to Hardware (ICES2000),

Lecture Notes in Computer Science, Vol. 1801, 252-263. Springer, Berlin.

[14] Vassilev, V. K. and Miller J. F. (2000) Scalability Problems of Digital Circuit Evolution, 2nd NASA/DOD Workshop on Evolvable Hardware, IEEE Computer Society Press, pp. 55-64.

[15] Walker, J. A. Miller, J. F. (2004) Evolution and Acquisition of Modules in Cartesian Genetic Programming, Proc. of the 7th European Conference on Genetic Programming, Lecture Notes in Computer Science, Vol. 3003, pp 187-197, Springer-Verlag, Berlin.

[16] Woodward, J. R. (2003) Modularity in Genetic Programming, Proceedings of the Fifth European Conference on Genetic Programming, Lecture Notes in Computer Science, Vol. 2610, pp. 258--267, Springer-Verlag, Berlin.

[17] Yu, T. and Miller, J. F. (2001) Neutrality and the Evolvability of Boolean Function Landscape, Proceedings of the 4th European Conference on Genetic Programming, Lecture Notes in Computer Science, Vol. 2038, pp. 204-217, Springer-Verlag, Berlin.