

The Automatic Acquisition, Evolution and Reuse of Modules in Cartesian Genetic Programming

James Alfred Walker and Julian Francis Miller

Abstract—This paper presents a generalization of the graph-based genetic programming (GP) technique known as Cartesian genetic programming (CGP). We have extended CGP by utilizing automatic module acquisition, evolution, and reuse. To benchmark the new technique, we have tested it on: various digital circuit problems, two symbolic regression problems, the lawnmower problem, and the hierarchical if-and-only-if problem. The results show the new modular method evolves solutions quicker than the original nonmodular method, and the speedup is more pronounced on larger problems. Also, the new modular method performs favorably when compared with other GP methods. Analysis of the evolved modules shows they often produce recognizable functions. Prospects for further improvements to the method are discussed.

Index Terms—Automatically defined functions (ADFs), Cartesian genetic programming (CGP), embedded Cartesian genetic programming (ECGP), genetic programming (GP), graph-based representations, modularity, module acquisition.

I. INTRODUCTION

SINCE THE introduction of genetic programming (GP) by Koza [1], researchers have been trying to improve the performance of GP and to develop new techniques for reducing the time taken to find an optimal solution to many different types of evolutionary problems. One such approach which was suggested by Koza and Rice, and has been widely adopted by the GP research community, is the inclusion of automatically defined functions (ADFs) in GP [2]. ADFs are reusable subroutines that are simultaneously evolved with the main GP program, and are capable of exploiting any modularity present in a problem to improve the performance of GP. An alternative approach to ADFs, called evolutionary module acquisition, is also capable of finding and preserving problem specific partial solutions in the genotype of an individual [3]. Since the introduction of these techniques, researchers have been interested in the potential and power that this feature brings to the evolutionary process. They have built on this work, or taken ideas from it, for their own specific situations, to reuse these partial solutions as functions elsewhere in the genotype [4]–[7].

Recently, another form of GP called Cartesian genetic programming (CGP), has been devised that uses directed graphs to represent programs, rather than the more familiar representation of programs as trees. Even though CGP did not have ADFs, it was shown to perform better than GP with ADFs on a number of problems [8], [9]. The work reported in this paper implements,

for the first time in CGP, a form of ADF based on the evolutionary module acquisition approach, which is capable of automatically acquiring and evolving modules. We call it embedded CGP (ECGP), as it is a representation that uses CGP to construct modules that can be called by the main CGP code. The number of inputs and outputs to a module are not explicitly defined but result from the application of module encapsulation and evolution.

The plan of this paper is as follows. Section II gives an overview of related work. In Section III, we describe the CGP technique, followed by our proposed extension to the technique, ECGP, in Section IV. Details of the experiment parameters are found in Section V. Sections VI–IX, give details of the even parity, adder, multiplier, and comparator experiments respectively, including the results and discussion. Section X describes the symbolic regression experiments and the results obtained. The lawnmower problem is described in Section XI, followed by the hierarchical if-and-only-if problem in Section XII. Section XIII gives our conclusions from the experimental findings and also some suggestions for further work.

II. MODULE ACQUISITION, MACROS AND AUTOMATICALLY DEFINED FUNCTIONS (ADFs)

When ADFs are used in GP, the process of finding a solution consists of the following stages: decomposing a problem into subproblems by defining a number of ADFs, solving each subproblem through the evolution of the ADFs, and then finally, assembling the solutions to the subproblems into a solution to the overall problem, through the reuse of ADFs in the main GP program [2]. Each ADF is defined in a separate *function defining* branch of the GP tree and requires the user to specify the number of arguments and the function set (which is allowed to contain ADFs) for each ADF. However, the output of each ADF is determined by evolution. The main GP program is defined in the *result-producing* branch of the GP tree and is only allowed to use ADFs defined in function defining branches of the same GP tree. The use of this form of ADF has been shown to increase the performance of GP on a number of problems [2].

The original idea of module acquisition [3] was to try and find a way of protecting partial solutions contained in the genotype, in the hope that it might be beneficial in finding a solution. The motivation for this idea arose because without module acquisition, desirable partial solutions may appear in the genotype but due to the stochastic nature of evolutionary operators, these partial solutions could be altered again into something less desirable. This could cause the evolutionary algorithm to take a longer amount of time to find a solution. Module acquisition

Manuscript received July 26, 2006; revised March 27, 2007.

The authors are with the Department of Electronics, University of York, Heslington, York YO10 5DD, U.K. (e-mail: jaw500@ohm.york.ac.uk; jfm@ohm.york.ac.uk)

Digital Object Identifier 10.1109/TEVC.2007.903549

attempted to preserve these partial solutions by introducing another two operators to the evolutionary process: *compress* and *expand*. The compress operator selects a section of the genotype and makes it immune to manipulation from evolutionary operators. This forms a module. The expand operator decompresses a module in the genotype, therefore allowing this section of the genotype to be manipulated by evolution once again. The fitness of a genotype is unaffected by these operators. However, these operators affect the possible offspring that might be generated using the evolutionary operators.

Atomization [3] not only makes sections of the genotype immune from manipulation by operators, but also represents the module as a new component in the genotype, thereby allowing the module to be manipulated further by additional compress operators. This allows the possibility of modules containing other modules, thus leading to a hierarchical organization 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) [5] also extracted program segments that were encapsulated and used to augment the GP function set. The system employed heuristics that tried to measure from population fitness statistics good program code, and also methods to detect when the search had reached local optima. In the latter case, the extracted functions could be modified. More recently however, Dessi *et al.* [4] showed that random selection of program subcode for reuse is more effective than other heuristics across a range of problems. Also, they concluded that, in practice, ARL does not produce highly modular solutions. It is important to note that once the contents of modules are themselves allowed to evolve, (as in this paper) they become a form of ADF. However, in contradistinction to Koza's form of ADFs [2] and Spector's automatically defined macros (ADMs) [6], there is no explicit specification of the number or internal structure of such modules. This freedom also exists in Spector's more recent PushGP [10].

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 [11]. 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 [12] showed that in general, the size of a solution is independent of the primitive function set used when modularity is permitted. This implied that including modules can remove any bias caused by the chosen primitive function set. Khare *et al.* also demonstrate that modular representations are more adaptable and evolvable than fully connected structures in dynamic environments [13].

III. CARTESIAN GENETIC PROGRAMMING (CGP)

Cartesian genetic programming (CGP) was originally developed by Miller and Thomson [9] for the purpose of evolving digital circuits. It represents a program as a directed graph (that for feedforward functions is acyclic). The benefit of this type of

representation is that it allows the implicit reuse of nodes, as a node can be connected to the output of any previous node in the graph, thereby allowing the repeated reuse of subgraphs. This is an advantage over tree-based GP representations (without ADFs) where identical subtrees have to be constructed independently. The CGP technique has some similarities with parallel distributed GP (PDGP), which was independently developed by Poli [14]. PDGP directly represents the graphs using a two-dimensional grid topology, in which each row of the grid is executed in parallel in the direction of data flow, with the program output being taken from the final row of the grid. This allows the formation of efficient programs by reusing partial results [14]. Originally, CGP also used a program topology defined by a rectangular grid of nodes with a user defined number of rows and columns. However, later work on CGP showed that it was more effective when the number of rows is chosen to be one [15]. This one-dimensional topology is used throughout the work we report in this paper.

In CGP, the genotype is a fixed length representation consisting of a list of integers which encode the function and connections of each node in the directed graph. However, CGP uses a genotype-phenotype mapping that does not require all of the nodes to be connected to each other, this results in the program (phenotype) being bounded but having variable length. Thus, there may be genes that are entirely inactive, having no influence on the phenotype, and hence the fitness. Such inactive genes therefore have a neutral effect on genotype fitness. This phenomenon is often referred to as neutrality. The influence of neutrality in CGP has been investigated in detail [9], [16], [15] and has been shown to be extremely beneficial to the efficiency of evolutionary process on a range of test problems.

In CGP, each node in the directed graph represents a particular function and is encoded by a number of genes. One gene encodes the function that the node represents, and the remaining genes encode where in the graph the node takes its inputs from. The nodes take their inputs in a feed forward manner from either the output of a previous node or from a program input (terminal). Also, the number of inputs that a node has, is dictated by the arity of the function it represents. The program inputs are labeled from 0 to $n - 1$, where n is the number of program inputs. The nodes in the genotype are also labeled sequentially starting from n to $n + m - 1$, where m is the user-determined upper bound of the number of nodes. The labels defined so far are used for referencing the program inputs and the outputs of the nodes. If the problem requires k program outputs, then k integers are added to the end of the genotype, each one encoding the output of the node where the program output is taken from. These k integers are initially chosen so that the program outputs are given by the outputs of the last k nodes in the genotype. Fig. 1 shows a CGP genotype and the corresponding phenotype that arose in the evolution of a 2-bit parallel multiplier. Fig. 2 shows how the CGP genotype is decoded to produce a phenotype.

A. The Efficiency of Cartesian Genetic Programming (CGP)

In its most common form (as used in this paper), CGP uses a very small population (five individuals). This is often evolved for a large number of generations, until a solution is found for a particular problem. This contrasts strongly

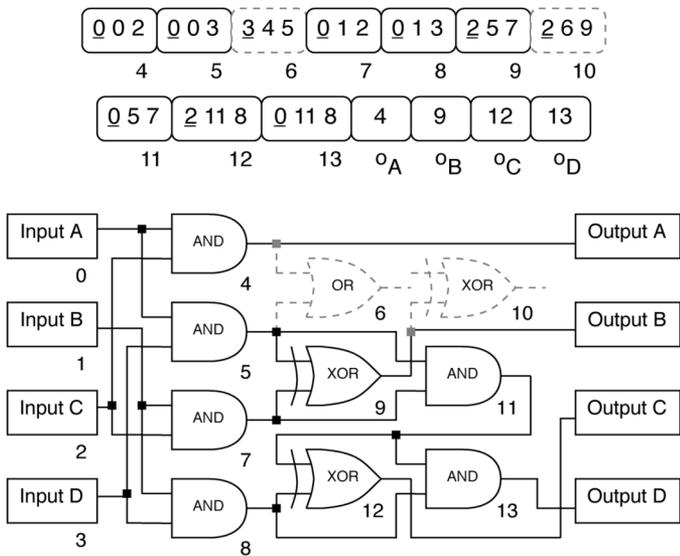


Fig. 1. A CGP genotype and corresponding phenotype for a 2-bit multiplier circuit. The underlined genes in the genotype encode the function of each node. The function lookup table is AND(0), AND with one input inverted(1), XOR(2), and OR(3). The index labels are shown underneath each program input and node in the genotype and phenotype. The inactive areas of the genotype and phenotype are shown in grey dashes (nodes 6 and 10).

with GP [1], [2], which starts with a very large population (possibly several thousand individuals), that is evolved until a solution is found to a particular problem, or a generation limit is reached (this is normally set quite low, possibly even 100 generations). However, when the number of fitness evaluations (number of generations \times population size) for the two different approaches are calculated, the difference between the efficiency of the two techniques is more comparable. On a number of benchmark problems, CGP has been shown to be more efficient than GP [9], [8].

Often CGP has been used to find efficient solutions to problems (i.e., in terms of the number of nodes used). This has been achieved by modifying the fitness function, so that when solutions are found they discriminated in favour of more efficient solutions. In these cases, the evolutionary algorithm is continued for a longer [17]. The disadvantage of this approach is deciding when to stop the evolutionary cycle, as quite often the smallest solution is unknown. Therefore, in these cases, an open-ended evolutionary process is left running for extremely long periods, to see if a more compact solution can be found. Another approach that was used to try to discover compact solutions restricted genotypes to have very short lengths. These were also evolved for a very large number of generations [18]. Although the latter approach succeeded in producing compact solutions, the very short genotypes hindered performance, as the amount of effort required to evolve solutions in CGP is strongly dependent on the chosen genotype length. This has recently been investigated in detail [19], where it was found that large genotypes required markedly less effort to evolve solutions. Both of these approaches were aimed at producing correct but also compact solutions.

It appears that there have been some misunderstandings in the literature regarding the efficiency of CGP, as some other re-

searchers have reported that CGP requires a very high number of fitness evaluations to find a solution to simple problems [20], [21]. This is not the case. There have also been other unfavorable comparisons between CGP and other techniques [22]. These have also been unfair as they have ignored not only the evolution speed versus genotype length issue but also have chosen different primitive functions sets (which invalidates comparison). From our experience, when CGP and other techniques (those cited above) are compared fairly (i.e., without the genotype length restriction in CGP), one finds that CGP is considerably more efficient than the reported alternatives. A useful set of computational effort figures for CGP that could be used for comparison appeared in 1999 [8], however, these have been ignored.

We offer the following guidelines for making fair comparisons in the future. First, make sure that the nature of the experiments you are comparing are the same (i.e., the explicit and implicit fitness functions are the same). Second, ensure that techniques with approximately equal program lengths are compared. Third, ensure that the primitive function sets are the same. Finally, take into account that thorough parameter sweeps are necessary, so that each technique's "sweet spot" is compared against the other's.

IV. EMBEDDED CARTESIAN GENETIC PROGRAMMING (ECGP)

ECGP is an enhancement of the CGP representation, that is being developed by Walker and Miller [23]. It incorporates some of the ideas from a technique known as module acquisition [3]. However, it extends the idea of automatic acquisition, and reuse of partial solutions (referred to as modules) by also allowing the evolution of partial solutions. It implements in addition an implicit selection pressure towards the formation of useful modules (i.e., ones that are associated with fitness improvement). In so doing, it brings to CGP a form of ADF [2].

A. Representation

The representation used in CGP had to be modified slightly in ECGP to allow the automatic acquisition and reuse of modules. In contrast to CGP, the ECGP *genotype* is a bounded variable length representation (in terms of the number of encoded nodes in the genotype and the number of genes used to encode each node). The number of nodes encoded in the genotype decreases when sections of the genotype are encapsulated into modules and increases when modules are expanded back into sections of the genotype. The number of genes used to encode the inputs of a node in the genotype can vary in two main ways. First, when a node in the genotype represents a module and the module mutation operators alter the number of inputs of the same module (thereby altering the number of genes required by the node in the genotype to encode the inputs of the module). Second, by the introduction of a module into the genotype by the compress or genotype point-mutation operators (so that the node representing the module in the genotype requires extra genes to encode all of the inputs of the module).

The way in which genes are represented has also been modified in ECGP. Each gene is now encoded using a *pair* of integers, rather than just a single integer as in CGP. This is illustrated in Fig. 3. One reason for this is that the modules are capable of having multiple outputs, and the CGP representation has hith-

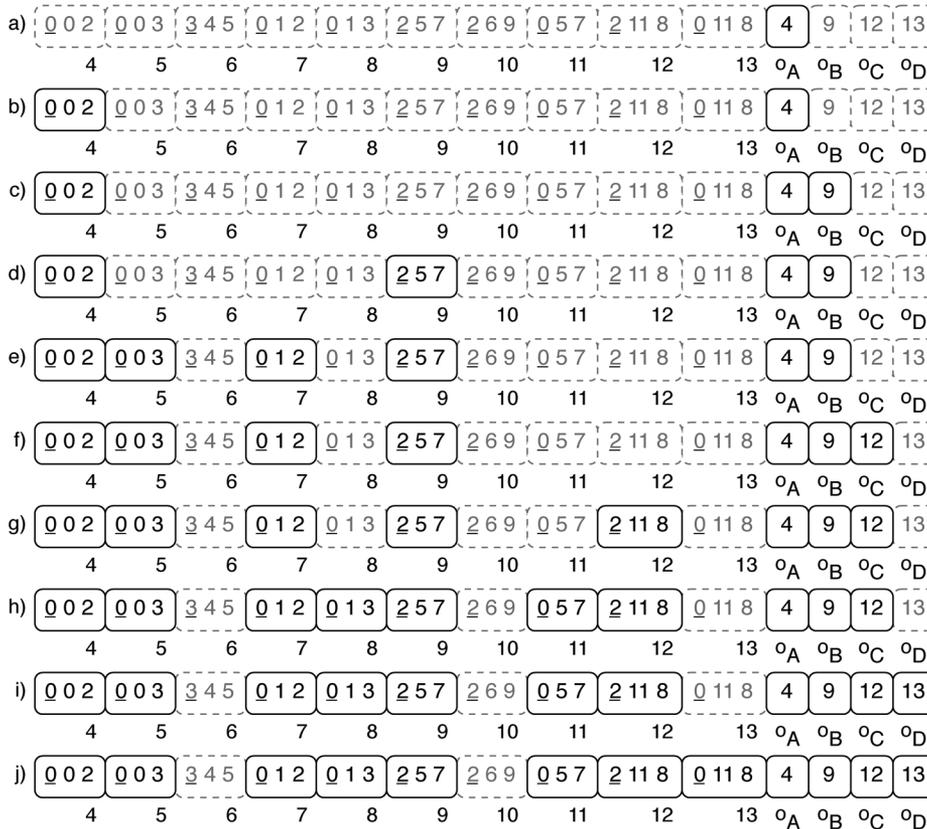


Fig. 2. The decoding procedure of a CGP genotype for the 2-bit multiplier problem. (a) Output A (o_A) connects to the output of node 4, move to node 4. (b) Node 4 connects to the program inputs 0 and 2, therefore, output A is decoded. Move to output B. (c) Output B (o_B) connects to the output of node 9, move to node 9. (d) Node 9 connects to the output of nodes 5 and 7, move to nodes 5 and 7. (e) Nodes 5 and 7 connect to the program inputs 0, 3, 1, and 2, therefore, output B is decoded. Move to output C. The procedure continues until output C (o_C) and output D (o_D) are decoded (steps (f) to (h) and steps (i) to (j), respectively). When all outputs are decoded, the genotype is fully decoded.

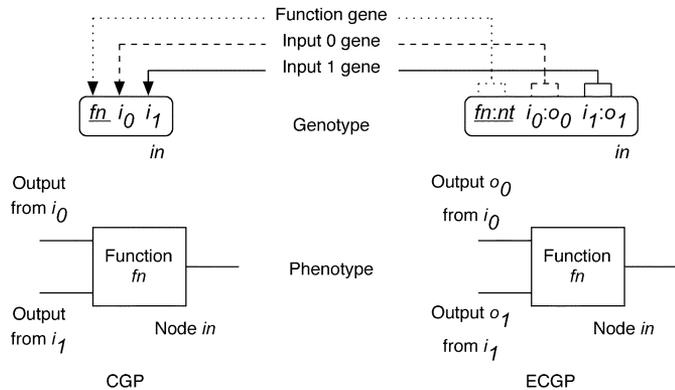


Fig. 3. Sections of CGP and ECGP genotypes encoding a single node and the corresponding phenotype for the node. In both cases, all of the genes are labeled. The separate components of each gene are also labeled as follows: function (fn), node type (nt), node indexes that the node inputs are taken from (i_0, i_1), node outputs that the node inputs are taken from (o_0, o_1), index of this node (in).

erto only encoded nodes representing functions with single outputs. For all of the node inputs encoded in the genotype, the first integer of each pair encodes the node index, just as it would in the CGP representation. However, the second integer of each pair encodes which output of a previous node the value is taken from (we emphasize, nodes in ECGP can have multiple outputs).

The other reason for encoding each gene using a pair of integers is to allow the introduction of node types into the ECGP representation. Node types allow the identification of three different kinds of nodes encoded in the genotype: primitive functions (node type 0), modules that contain an original section of the genotype (node type I), in which modules are not present, and finally, modules that have been reused (node type II), therefore containing a replicated section of the genotype. Different node types need to be identified in the genotype, as the evolutionary operators act differently on the nodes depending on their node type (this is explained further in Section IV-C). Node types are encoded in the genotype as the second integer of each pair which encode the function gene of each node, the first integer encodes the primitive function (as it would in CGP) or module the node represents (using values from a lookup table). An example of CGP and ECGP genotypes that encode the same phenotype is shown in Fig. 4 to illustrate the differences between the representations.

B. Module Representation

A module is represented as a bounded variable length genotype, which has the same characteristics as an ECGP genotype, except the number of nodes represented in the module genotype remains fixed. However, the number of module outputs encoded in the module genotype can vary. The module genotype consists

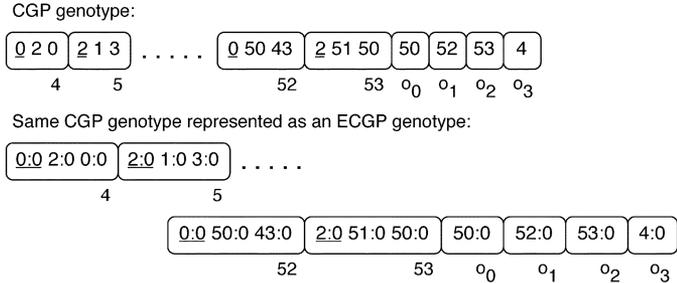


Fig. 4. Examples of CGP and ECGP genotypes encoding the same phenotype for a 2-bit multiplier circuit (four inputs, four outputs). For each encoded ECGP node, the underlined pair of integers encode the function and the node type, the remaining pairs of integers encode the node inputs. Every node encoded in this 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.

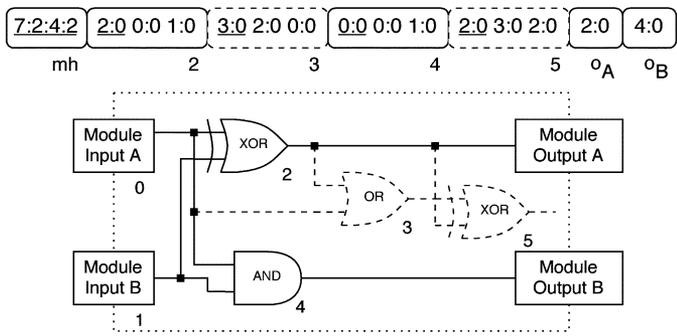


Fig. 5. The genotype and corresponding phenotype of a module constructing a half adder circuit. The first section of the genotype is the module header (mh). For each node, the underlined genes encode the function, the remaining genes encode the node inputs. The function lookup table is AND(0), XOR(2), and OR(3). 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.

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. These four integers encode respectively, 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 standard ECGP genotype. An example of a module genotype showing the separate components is shown in Fig. 5, along with its corresponding phenotype.

The size of a module genotype is determined by the number of nodes and module outputs 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, which is set by the user. Also, the number of module outputs encoded in the module genotype is bounded between a minimum limit of one (otherwise, there would be no way to connect to the module and access its result) 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 a module is allowed to have is also restricted between a minimum of two and a

maximum of $2n$ module inputs. However, the number of module inputs allowed does not affect the size of the module genotype, as the module inputs 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. This is to prevent the module point-mutation operator from modifying existing module genotypes, and the compress operator from creating new module genotypes, in such a way as to create multiple-level nested modules within other modules. The excessive nesting of modules in turn leads to excessive code growth through the reuse of modules and eventually causes implementation issues such as “stack overflow” and “out of memory” errors when decoding the individual genotypes. In future work, methods to prevent these implementation problems while allowing modules within modules will be investigated.

Not all of the nodes outputs have to be connected in the module genotype. This leads to a form of neutrality, identical to that found in CGP, to exist inside the modules. This can be seen in Fig. 5, where nodes 3 and 5 are not connected. The contents of a module 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 mutation operator see Section IV-D).

Once a module is created, the module genotype is stored in a global *module list*, which is an extension of the primitive function list and is shared by all individuals in the population. Any node in a genotype can be mutated to represent any module or primitive function present in either list (providing the rules for node type are obeyed. See Section IV-C for more information on node types). The module list is dynamic and has no restrictions on its maximum size. When the fittest individual of the population is promoted to the next generation (chosen in accordance with the evolutionary strategy used in Section V-A), the module list is updated to include only those modules present in the fittest individual, thereby deleting all modules present in the module list that were only found in less fit individuals of the population. It was found that this creates a regulatory control of the module list and prevents excessive growth of the module list occurring.

C. Genotype Operators and Node Types

ECGP extends CGP by allowing the use of dynamic acquisition, evolution and the reuse of modules. This is achieved through extra mutation operators, which are used in conjunction with the genotype point mutation of CGP.

1) *Compress and Expand*: The compress operator constructs modules by selecting two random points in the genotype (in accordance with the module size limits) and encapsulates all the nodes (of node type 0) between these two points into a new module. This is encoded into a module genotype of the form described earlier, in Section IV-B. If there are any nodes of type I or type 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 that a module is initialized with is determined by the number of connections between the inputs of the nodes which are going to be encapsulated into a module, and the outputs of any previous

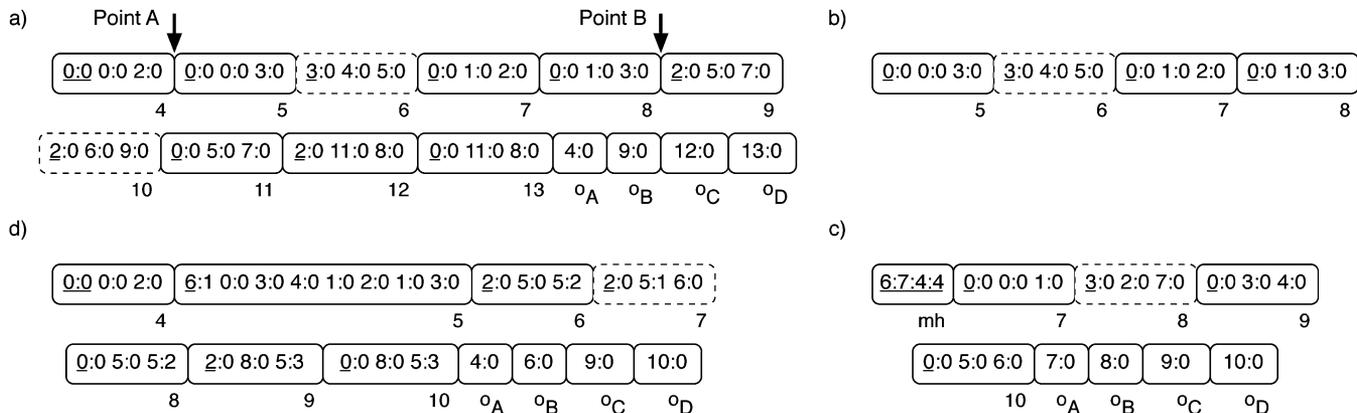


Fig. 6. The four steps of the compress operator. (a) A section of the genotype is chosen by randomly choosing points A and B. (b) The chosen section is removed from the genotype, (c) The chosen section is converted into a module genotype by adding a module header and four module outputs. All nodes and their inputs are also relabeled. (d) A node of type I representing the new module is added to the genotype in place of the removed section. All nodes, node inputs, and outputs occurring after the inserted node are relabeled.

nodes or program inputs (terminals) in the genotype when the module is created. If there are repeated connections to the output of a previous node, each connection is assigned its own module input. Likewise, the number of module outputs possessed by a module is determined by the number of connections between the inputs of the later nodes in the genotype (to the right of the right-hand module boundary) 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. An example of the compress operator creating a module in the genotype is shown in Fig. 6. The function gene of any type I node is immune from the genotype point mutation operator. A type I node can only be removed by the expand operator.

The expand operator randomly selects a type I node in the genotype, and replaces the type I node with the nodes contained in the module represented by the type I node. This operation can only be applied to type I nodes, as the nodes contained inside a module represented by a type I node were originally part of the genotype. An example of the expand operator is the reverse of Fig. 6.

The inputs of all of the later nodes in the genotype of the individual are updated in the final stage of both the compress and expand operators, so all of the connections remain intact. The compress and expand operators only make a structural change to the genotype and have no effect 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. The expand operator can be looked at as a module destruction operator. The only way that modules can replicate in a genotype is through the action of the genotype point mutation operator, which changes the function gene of a node into a value that represents a module, thereby performing a module duplication. Since an evolutionary strategy is used that promotes the single fittest genotype to the next generation, the only way that modules can survive in the genotype is by being present in this promoted genotype. We found that this introduces an implicit pressure for good modules to replicate quickly in the genotype

in order to survive. Thus, the modules within the genotype undergo a struggle for survival. This is in addition to the usual fitness-based selection that is applied to the genotypes.

2) *Genotype Point Mutation*: Modules can replicate within the genotype through the action of the genotype point mutation operator. This is identical to the point mutation operator used in CGP, with the exception that it can mutate the function of a node to represent any of the primitive functions or available modules in the module list.

If the function gene of a type 0 node (a node representing a primitive function) is mutated to represent a module, it is classed as a type II node. The genotype point mutation operator treats type 0 and type II nodes the same way. Therefore, the genotype point mutation operator can also mutate the function gene of a type II node to represent any of the predefined functions (thereafter referred to as a type 0 node) or available modules in the module list (still referred to as a type II node). Whenever a node changes type (from type 0 to type II, or *vice versa*), the new node keeps however many of the original node's inputs it requires, and "randomly" generates any extra inputs it may require. This may also happen when a type II node representing module A is mutated to represent module B, when module B has a different number of module inputs than module A.

The genotype point mutation operator can also mutate the inputs of any type 0, type I, or type II node. The only difference from the CGP point mutation operator is that in ECGP every input gene has two values which need to be mutated, one for the node index and one for the node output. Both of these are mutated at the same time, to ensure that every connection in the graph is still valid.

Type II nodes are also immune from the expand operator. This was introduced to avoid the excessive growth of the genotype that could occur when type 0 nodes were replicated to form type II nodes that in turn were expanded back into the genotype.

To summarize, the properties of type 0, I, and II nodes are shown in Table I, with the affect the genotype operators have on them. The reason for having type I and II nodes is to try and reduce the excessive growth of the genotype, and to help induce a selection pressure on the modules. The modules have to

TABLE I
THE EFFECT OF THE OPERATORS ON EACH NODE TYPE

Node Type	Action of Compress	Action of Expand	Action of Genotype Point Mutation
0	Encapsulates into a module	Immune	Changes function or node inputs
I	Immune	Replaces with module contents	Changes node inputs
II	Immune	Immune	Changes function or node inputs

replicate in the genotype (i.e., make the transition from being represented by type I to type II nodes) 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 that it will be removed from the genotype (i.e., it cannot be expanded). This is advantageous, as it allows good modules to stay in the module list. However, it is also disadvantageous, as it could possibly allow the evolution of the genotype to progress at a slower rate, as it is harder to remove modules which contribute towards convergence on local optima.

D. Module Operators

The module genotypes contained in the module list can also be evolved through the action of five different module mutation operators: module point mutation, add-input, add-output, remove-input, and remove-output. All of the module mutation operators must comply with the restrictions on the number of module inputs and the number of module outputs at all times.

1) *Module Point Mutation*: The module point mutation operator is a restricted version of the ECGP genotype point mutation operator, as it can still mutate the input and function genes of any node encoded in the module genotype. However, it is not allowed to introduce any type II nodes into the module genotype. Therefore, it can only mutate the function of a node to represent one of the predefined primitive functions. The module point mutation operator can also mutate which node outputs the module outputs are connected to. However, a module output can never be mutated to connect directly to a module input, as this would bypass any processing the nodes in the module may perform on the data presented at the module inputs (i.e., it would implement a “junk” module).

2) *Add-Input and Add-Output*: The add-input and add-output operators allow greater connectivity to and from the nodes contained in a module, by increasing the number of module inputs or module outputs by one respectively, each time either operator is applied (providing the constraints for the number of module inputs and module outputs a module can possess are obeyed). 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. An extra gene is also inserted into all nodes (type I and type II) representing the module in the genotype (to update the arity of the node). The extra gene is randomly assigned a 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. An extra gene is also added to the end of the module genotype. The extra gene is randomly assigned values for the node index and node output, which encode where the new module output is connected to. Examples of the affect the add-input and add-output operators have on the module genotype and individual genotype are shown in Fig. 7.

3) *Remove-Input and Remove-Output*: 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 (once again making sure to obey the constraints imposed on a module regarding the number of module inputs and module outputs that it may have). 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 randomly chosen module input 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 genotype.

V. EXPERIMENT DETAILS

All of the experiments reported here were run on a 2.2 GHz single processor desktop PC with 448 MB of memory. The time taken to complete 50 runs of each problem varied between a few seconds and 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 associated with module acquisition is quite small, and the computational time taken for fitness evaluation (both CGP and ECGP) is by far the dominant factor.

A. Evolutionary Strategy

Both CGP and ECGP used the $(1 + 4)$ evolutionary strategy [24] defined as follows.

- 1) Randomly generate an initial population of five genotypes and select the fittest.
- 2) Carry out mutation on the winning parent to generate four 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.

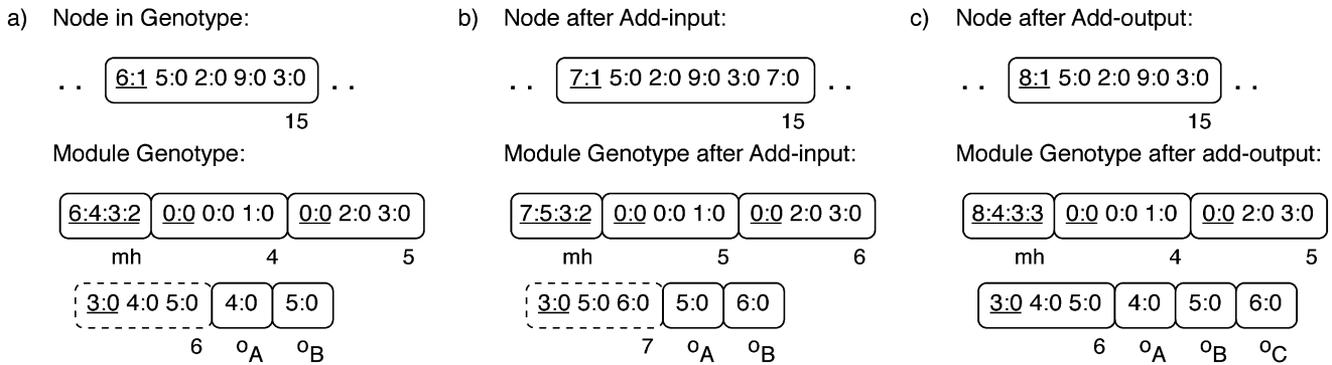


Fig. 7. The add-input and add-output operator. (a) A node representing module 6 in a genotype and the corresponding module genotype for module 6. (b) The node and module genotype after the add-input operator has been applied to (a). The number of inputs gene in the module header (mh) of the module genotype has been incremented by one and the module identifier has been changed to the next available number. The function of the node has been changed to reflect the new module identifier and an extra gene has been added for the new module input, whose value is randomly chosen. (c) The node and module genotype after the add-output operator has been applied to (a). The number of outputs gene in the module header (mh) of the module genotype is incremented by one and an extra gene is added to the module genotype to encode the new module output, whose value is randomly chosen. The module identifier is also changed to the next available number. The function of the node is also updated to reflect the change in the module identifier.

TABLE II
THE COMMON PARAMETERS USED FOR CGP AND ECGP ON ALL
OF THE TEST PROBLEMS. “*” DENOTES ECGP ONLY

Parameter	Value
Population size	5
Initial genotype size	100 nodes (300 genes)
Genotype point mutation rate	3% (9 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
Module list initial contents *	Empty
Number of independent runs	50

- 5) Go to Step 2 unless the maximum number of generations is reached or a solution is found

In Step 4b of the evolutionary strategy, the offspring is chosen over the parent when they both have an equal fitness, as the offspring is phenotypically identical (in terms of fitness), but genetically different from the parent. Thereby allowing neutral exploration of the search space until a phenotypically better offspring is discovered.

B. Parameters

The parameters used for CGP and ECGP which are common to all of the experiments are shown in Table II. The operator rates and probabilities were determined to be fairly optimal by means of a trial and error process in previous work [23], [25], [26]. For all experiments, a maximum number of generations is not set for each independent run. Instead, each independent run continues until a solution is found.

C. Performance Statistics

In each experiment, the results for all independent runs were assessed using a statistic called “computational effort.” This metric was introduced by Koza in [1], as a measure of the computational effort required to solve a problem based on the data from all of the independent runs. The formula to calculate computational effort is shown in (1). The notation is taken from [1] as follows: $N_s(i)$ —the number of successful independent runs by generation i , N_{total} —the total number of independent runs, $P(M, i)$ —the cumulative probability of success for an independent run with population size M producing a solution by generation i , $R(z)$ —the number of independent runs required to satisfy the success predicate by generation i with probability z , $I(M, i, z)$ —the number of individuals that need to be processed to produce a solution with probability z , using population size M , at generation i . In this paper, we use $z = 0.99$

$$P(M, i) = \frac{N_s(i)}{N_{total}}$$

$$R(z) = \text{ceil} \left(\frac{\log(1-z)}{\log(1-P(M, i))} \right)$$

$$I(M, i, z) = MR(z)i + 1$$

$$CE = \min_i I(M, i, z). \quad (1)$$

The computational effort statistic was used in this paper as it is the most commonly used performance measure in the GP community. However, it is by no means perfect and has numerous inadequacies. Christensen and Oppacher [27] found that the *ceiling* operator in (1) has a tendency to overestimate $R(z)$, while the *min* operator tends to underestimate the computational effort required. Furthermore, the underestimation increases in systems with a high number of generations, which is the case in the approach used in this paper. Niehaus and Banzhaf [28] later found that the underestimation of the computational effort statistic is inversely proportional to the number of runs used in the calculation, so for a small number of runs, the underestimation of computational effort is very large. In this paper, only 50 independent runs are used (which is classed as a small number of runs) for each experiment, as this was the number of runs

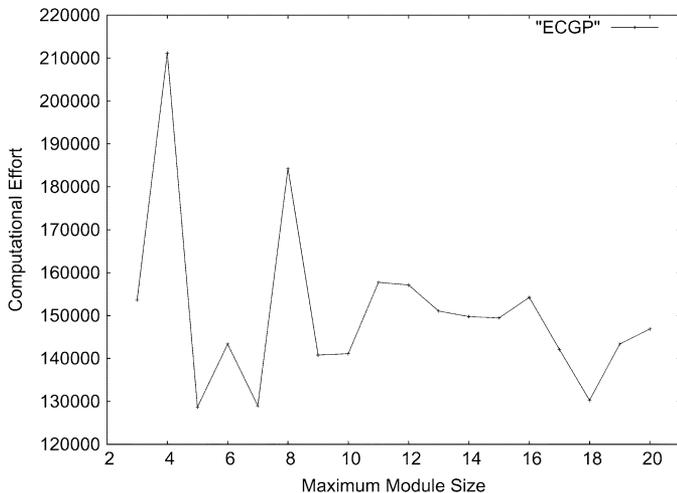


Fig. 8. A comparison between the maximum module size and the performance of ECGP in terms of CE for the even four parity problem.

usually used in the work we compare with. Therefore, the computational effort figures are likely to be underestimates of the theoretical value for computational effort and should only be used as a rough guide. However, Niehaus and Banzhaf [28] also found that as the probability of a run ending in failure increased, the computational effort deviated further from the theoretical value. In this paper, every run continues until a solution is found, thereby producing a 100% success rate, which should improve the accuracy of the computational effort values.

Since there are still questions concerning the accuracy of the computational effort statistic, we have also compiled a variety of other statistics: median number of evaluations, median absolute deviation, and interquartile range for both CGP and ECGP. The results from all of the experiments in this paper are positively skewed (since the minimum number of evaluations is 1), so the mean and standard deviation statistics are not used, as they would not provide an accurate and meaningful representation of the data. We have assessed the significance of the results using the nonparametric Mann–Whitney U test [29], as parametric significance tests (such as the t -test [30]) can only be used on data which is normally distributed. As suggested by Christensen and Oppacher, the CGP and ECGP data sets collected from all runs will be made available from the CGP website,¹ so authors can compare with these figures in the future (in addition, source code for these experiments will also be made available).

D. What Is a Suitable Maximum Module Size?

In ECGP, the user sets a parameter, ms , for the maximum module size. This allows ECGP to construct modules of size n , where $2 \leq n \leq ms$. The main concern is how to determine a good maximum module size. To shed light on this issue, we have applied ECGP to the even 4 parity problem (also used in Section VI), using the parameters in Table II and various maximum modules sizes. The aim of the investigation was to see if any correlation exists between the maximum module size and the performance of ECGP. The results are shown in Fig. 8. It can

¹The CGP website is currently under construction and can be found at <http://www.cartesiangp.co.uk>

be seen that there is no obvious correlation between the maximum module size and computational effort. However, there is a weak trend that implies that larger module sizes improve the performance of ECGP. One possible reason to explain this is that larger modules contain more inactive nodes, so that neutrality (the same type as in CGP) could be having an impact. Another possible reason is, that when larger modules are created there must be fewer in number. This is due to modules not being permitted within modules, thus implying that the existing modules are evolved for a longer period of time. All of these aspects will be investigated further in future work. For the moment, it appears the best maximum module sizes for the even 4 parity problem are 5, 7, and 18. In this paper, we used a maximum module size of five for most of the experiments, as this is the smallest and will allow faster evaluation of the evolved programs.

VI. EVEN PARITY PROBLEM

The problem of evolving even parity functions using GP with the primitive Boolean operations of AND, OR, NAND, NOR has been shown to be very difficult and has been adopted by the GP research community as a good benchmark problem for testing the efficacy of new GP techniques [1]. It is particularly appropriate for testing module acquisition techniques, as even-parity functions are more compactly represented using XOR and XNOR functions. Also, smaller parity functions can help build larger parity functions. Thus, parity functions are naturally modular, and it is to be expected that they will be evolved more efficiently when such modules are provided. It is, therefore, of great interest to see whether modules that represent such functions are constructed automatically.

The even n parity problem has n inputs and a single output, which produces a 1 if there are an even number of 1's in the inputs, and 0, otherwise. In this paper, CGP and ECGP are applied to the even n parity problem, where $n = 3, 4, 5, 6, 7, 8$. The function set used when evolving the even parity problem consisted of the Boolean functions: AND, NAND, OR, and NOR. The maximum module size used for ECGP is five. Both CGP and ECGP use a genotype of 50 nodes.

A. Results and Discussion

The computational effort (CE) figures for CGP and ECGP applied to the even n parity problem, where $n = 3, 4, 5, 6, 7, 8$, are shown in Table III.

In all 50 runs, both CGP and ECGP produced 100% successful solutions. As the complexity of the problem increases, it can be seen that ECGP performs significantly better than CGP, and the difference in performance between ECGP and CGP increases. This suggests that ECGP may perform even better than CGP on more complex problems. It appears that the performance increase found in ECGP is due to the discovery, preservation, and reuse of partial solutions in the genotype, whereas CGP has to find each partial solutions separately. For the low order parity functions, CGP performs better than ECGP. In this case, it is likely that the overhead of discovering useful modules and learning how to reuse the modules, is responsible for the decrease in performance.

TABLE III
THE COMPUTATIONAL EFFORT FIGURES FOR CGP, ECGP, GP, AND EP (BOTH WITH AND WITHOUT ADFs) APPLIED TO VARIOUS SIZE EVEN PARITY PROBLEMS. THE COMPUTATIONAL EFFORT FIGURES FOR GP AND EP ARE TAKEN FROM [2] AND [31]. “**” DENOTES COMPUTATIONAL EFFORT FIGURES CALCULATED WITH $z = 1.0$

Parity	CGP	ECGP	GP	GP with ADFs	EP	EP with ADFs
3	33,282	37,446	96,000	64,000	28,500 *	63,000
4	151,683	201,602	384,000	176,000	181,500	118,500 *
5	776,002	512,002	6,528,000	464,000	2,100,000	126,000
6	3,044,162	978,882	70,176,000	1,344,000	-	121,000 *
7	11,451,202	1,923,842	-	-	-	169,000 *
8	31,187,842	4,032,002	-	-	-	321,000

A closer examination of the solutions found by ECGP, revealed that a high percentage of the nodes contained in the genotype represented modules. Also, the majority of the modules have been reused repeatedly. Further inspection of the modules showed that a large number of them represent either an XOR or XNOR function. Some of the evolved XOR and XNOR functions were represented in their most compact form, consisting of three Boolean functions. Other XOR and XNOR functions were represented in a much less efficient form, consisting of up to five Boolean functions. The majority of the modules also contained some inactive nodes, indicating that neutrality also occurs in the module genotype, while the modules are being evolved.

Comparing the results of CGP and ECGP with GP, for the even n parity problem, where $n = 3, 4, 5, 6$, shows CGP and ECGP significantly outperforming GP and the speedup increasing with problem complexity. However, GP has been reported to be unable to find any solutions on the higher order parity functions [2]. Comparing ECGP and GP with ADFs, shows a similar trend for all of the even parity problems tested. This is an interesting result, as GP allows a two level hierarchy in the ADFs, thereby allowing the creation of subparity functions. ECGP, however, only allows a single-level hierarchy in its modules, where only the XOR or XNOR function can be created. Therefore, the most likely cause of the performance increase between CGP, ECGP and GP, is the implicit reuse of nodes found in the directed graph representation. In addition to the results in Table III, GP with ADFs is capable of finding solutions to higher order parity problems. However, different ADF parameter settings are used for the runs, which allowed the ADFs to have a larger number of arguments, therefore making an unfair comparison with the rest of the results. When a greater number of arguments are allowed in ADFs, larger subparity functions can be created, causing an increase in performance [2]. This suggests that allowing a variable number of module inputs in ECGP in conjunction with larger module sizes could provide further performance increases in ECGP.

The computational figures for evolutionary programming (EP) [31], with and without ADFs, also listed in Table III show a strange correlation, as EP with ADFs finds the even 6 parity problem easier to solve than the even 5 parity problem, when the opposite should be true. Also, some of the figures for EP are computed with $z = 1.0$, making the computational effort

formula invalid and comparisons more difficult. However, comparing the EP figures with CGP and ECGP shows the results for CGP and ECGP scale much better with problem difficulty than EP. When CGP and ECGP are compared against EP with ADFs, EP with ADFs is shown to perform much better than CGP and ECGP on all but the smallest parity problem. The ADFs in EP also use a multilevel hierarchy, like those found in GP. However, they seem to have a more prolific effect on the performance of EP than GP, which could possibly be attributed to some of the extra mutation operators used in EP. As previously mentioned, ECGP only uses a single-level hierarchy, which in this experiment could only construct either the XOR or XNOR function, whereas EP, like GP, could construct various subparity functions. This again strongly suggests that when a multilevel hierarchy is allowed in ECGP, a significant improvement in performance could be possible.

As a further investigation to see if allowing larger modules capable of encoding smaller parity functions would improve performance, we carried out the following experiments. ECGP was rerun on the even parity problem, where $n = 4, 5$, with a maximum module size of 20 nodes and a genotype size of 400 nodes (keeping the same maximum module size to genotype size ratio as the previous experiments). The extra resources provided a significant performance boost in ECGP. The computational effort for the even 4 parity was 32 641, a speedup of 6.18 compared with the original ECGP result for the same problem, which is also significantly better than EP with ADFs. For the even 5 parity, the results were not as impressive but still comparable to the result of EP with ADFs, as the computational effort figure of ECGP was 130 081. This is still an improvement on the original results for ECGP by 3.94 times. However, all of the computational effort figures for CGP and ECGP are heavily dependent on the number of nodes encoded in the genotype [19]. This can cause vast fluctuations in the performance of both techniques, thereby making fair comparisons with other techniques very difficult, especially with nongraph based approaches, where it is difficult to quantify the number of resources used.

The improvement in performance of ECGP in the further experiments, was due to modules being allowed to construct multiple XOR or XNOR functions inside a single module, thereby forming larger subparity functions than in the previous run of ECGP. However, the limit on module resources and the single-

TABLE IV
THE MEDIAN NUMBER OF EVALUATIONS (ME), MEDIAN ABSOLUTE DEVIATION (MAD), AND INTERQUARTILE RANGE (IQR) OF CGP AND ECGP FOR VARIOUS SIZE EVEN PARITY PROBLEMS. THE U VALUE IS FROM THE MANN-WHITNEY SIGNIFICANCE TEST AND IS HIGHLY SIGNIFICANT ($P < 0.001$) WHEN DENOTED BY[‡]

Parity	CGP			ECGP			U
	ME	MAD	IQR	ME	MAD	IQR	
3	5,993	2,936	6,610	5,931	3,804	10,372	1,299
4	30,589	12,942	25,438	37,961	21,124	49,552	1,521
5	136,693	71,236	199,245	119,625	52,483	98,940	1,128
6	577,237	257,574	594,770	227,891	85,794	190,456	449 [‡]
7	2,156,139	1,029,010	2,343,039	472,227	312,716	603,643	185 [‡]
8	7,166,369	3,210,224	6,363,465	745,549	500,924	1,108,934	39 [‡]

level hierarchy still remains, which explains why the speedup decreases between the even 5 and even 6 parity results, and why ECGP still performs worse than EP with ADFs in both cases. From these results, it is possible to say that the introduction of a multilevel module hierarchy could be beneficial to the performance of ECGP, as it would be less dependent on module resources to build larger functions.

Spector and Robinson’s PushGP system [32] has been applied to the even n parity problem, where $n = 3, 4, 5, 6$. PushGP uses a standard GP representation to evolve a set of instructions (including various stack-manipulation instructions, subroutines and control structures), which are then executed on a combination of global stacks (of various data types) containing the arguments (inputs) associated with the problem. Modularity emerges in PushGP through the execution of instructions on the stacks rather than through the structure of the representation. However, due to the nature of PushGP system and its instructions set, direct comparisons could not be made with GP (with or without ADFs), as PushGP was seen to have a “considerable advantage” over GP [32]. Therefore, direct comparisons cannot be made with CGP and ECGP either.

Poli and Page have also applied their work using a smooth uniform crossover in GP to the even parity problem [33]. However, we are unable to compare with this approach as a different function set was used. This affects the difficulty of the even parity problem and would lead to an unfair comparison between the techniques.

Another GP technique, known as enzyme GP [22], has also been applied to the even 3 and even 4 parity problems. Enzyme GP uses a biomimetic representation based on an “enzyme system.” The representation defines the functional elements of the system (i.e., primitive function set) but the phenotype arises through interactions between the functional elements and can be evaluated like a GP tree. Also, functional elements are allowed to be reused, thereby allowing the exploitation of modularity. For the even 3 and even 4 parity problems, enzyme GP has a computational effort of 79 000 and 2 588 750, respectively. CGP and ECGP perform significantly better than enzyme GP on both parity problems. In fact, CGP performs 2.37 and 17.07 times

faster than enzyme GP, while ECGP performs 2.11 and 12.84 times faster than enzyme GP for the even 3 and even 4 parity problems, respectively.

In addition to the computational figures in Tables III, Table IV shows the median number of evaluations required by CGP and ECGP to find a solution to each of the evolve parity problems. Both tables show a similar trend for the performance of CGP compared with ECGP. An interesting observation from Table IV is that the median absolute deviation of ECGP becomes much lower than that of CGP, as the parity problem increases in difficulty. This indicates that the use of modules in ECGP improves the accuracy of the technique as well. However, the figures for the Mann-Whitney U test shows that there is no significant difference between CGP and ECGP on the 3-, 4-, and 5-bit even parity problems but the difference between CGP and ECGP on the 6-, 7-, and 8-bit even parity problems is highly significant ($P < 0.001$). This is an encouraging result, as it further supports the hypothesis that the automatic acquisition, evolution and reuse of modules improves the performance of the technique.

VII. DIGITAL ADDER PROBLEM

The digital adder is a more recent problem, which has emerged from research in the field of Evolvable Hardware. It is now a commonly used benchmark for GP techniques with multiple outputs. It is considerably harder and more complex than the even parity problem, as the digital adder problem has multiple outputs. Also, the number of inputs increases much faster than the even parity problem, as the complexity of the test problem is increased. A n -bit digital adder has two n -bit inputs, and a 1-bit carry-in, giving a total of $2n + 1$ inputs. Likewise, it also has a n -bit sum output, and a 1-bit carry-out, giving a total of $n + 1$ outputs.

The function set used is identical to the even parity problem (AND, NAND, OR, NOR). The function set was chosen because the digital adder is also a modular problem. The digital adder is constructed more compactly when an XOR function is allowed, thus the situation is reminiscent of the even parity problem. The maximum module size for ECGP was once again set to five nodes.

TABLE V
THE COMPUTATIONAL EFFORT FIGURES FOR CGP AND ECGP APPLIED
TO THE 1-BIT, 2-BIT, AND 3-BIT DIGITAL ADDER PROBLEMS

Adder	CGP	ECGP	Speed-up
1	36,486	43,203	0.84
2	834,246	596,643	1.40
3	8,599,682	3,414,723	2.52

A. Results and Discussion

Computational effort figures for the results of CGP and ECGP on the 1-, 2-, and 3-bit digital adder problems were calculated using the formula in (1) and are shown in Table V. Unfortunately, GP researchers tend to avoid problems with multiple outputs, therefore we have no figures for GP applied to the adder problem to compare with those of CGP and ECGP. However, computational effort figures have been published for Enzyme GP applied to the 2-bit adder, but a function set is used which makes the problem easier to solve, as no intermediate building blocks (such as the XOR function) have to be constructed. Therefore, we cannot use these figures for a fair comparison.

All of the 50 runs produced 100% successful solutions of both CGP and ECGP. The results are similar in nature to those found with even parity problems. The 1-bit adder (which is by far the least complex of the adders to evolve) is evolved quicker using CGP than with ECGP. Once again, this could be accounted for by the overhead of the exploration of code in the modules, and also ECGP not being able to reuse the discovered modules correctly. On the harder 2- and 3-bit adders, it can be seen that ECGP starts to outperform CGP. Also, as the problem scales in complexity, the speedup between CGP and ECGP increases as well. This can be attributed to ECGP finding useful partial solutions, and then reusing them throughout the genotype, exploiting any modularity found in the adder.

On closer inspection of the evolved modules, it can be seen that the modules mainly represent the XOR and the half adder functions, which are both partial solutions for the adder problem. In a similar manner to the evolved modules for the even parity problem (see Section VI), we found that the XOR and half adder functions were constructed in various ways, and varied greatly in complexity. Also, the majority of nodes in the genotype of an ECGP individual encoding a solution, represented modules rather than primitive functions. This implies the modules are more appealing than the primitive functions, in terms of obtaining a high fitness score. The same phenomenon was also seen in the solutions to the even parity problem.

From Table VI, the figures for the median number of evaluations of CGP and ECGP follow a similar trend to those of the computational effort figures in Table V. Also, the degree of variance in the results for ECGP is lower than that of CGP, as the difficulty of the problem increases, indicating the ECGP figures could be more accurate than those of CGP. Table VI also shows that only the performance difference between CGP and ECGP on the 3-bit adder is highly significant, indicating that ECGP seems to perform better on harder problems than CGP.

VIII. DIGITAL MULTIPLIER PROBLEM

The digital multiplier problem is by far the hardest of the digital circuits evolved in this paper. Some researchers (especially in the Evolvable Hardware field) have now adopted the multiplier problem as a benchmark, for testing new techniques (in addition to the digital adder) and new approaches for speeding up the evolutionary process (such as decomposing problems into subproblems or using a subset of the truth table). The largest multiplier evolved so far has been the 5-bit digital multiplier [34].

The n -bit digital multiplier takes two n -bit numbers as its inputs, and multiplies the numbers together, to produce a $2n$ -bit number as its output. In this paper, CGP and ECGP are applied to the 2- and 3-bit multiplier problems (the 1-bit multiplier is simply the AND function). In conventional human design, the multiplier is constructed using a series of full and half adders in a lattice type structure, with the carry out of each adder being propagated through the lattice. In this paper, the function set used in CGP and ECGP consisted of the Boolean functions AND, AND with one input inverted, XOR and OR. The experiment would be considerably easier if the half adder and the full adder were included in the function set, as these functions allow the circuit to be represented more compactly. However, both of these functions can be constructed from the function set, as the multiplier problem is also naturally modular. In addition to the parameters in Table II, we chose a maximum module size of five.

A. Results and Discussion

Computational effort figures for the results of CGP and ECGP applied to the 2- and 3-bit multiplier problems are shown in Table VII. GP researchers tend to avoid the multiplier problem, due to its multiple outputs. Therefore, we have no figures for GP to compare with those of CGP and ECGP. However, a comparison is made with enzyme GP on the 2-bit multiplier problem.

Once again, all 50 independent runs of CGP and ECGP produced 100% successful solutions for both multiplier problems. The results show ECGP performs 4.84 times faster than CGP on the harder 3-bit multiplier problem, but only performs at 0.69 times the speed of CGP on the simpler 2-bit multiplier problem. The statistics in Table VIII also follow a similar trend in the performance to the computational effort figures of CGP and ECGP and show that on the larger 3-bit multiplier problem, the result is highly significant. Supporting the results for other problems in this paper, this result suggests the overhead of automatic acquisition and reuse of modules on simple problems is detrimental to the performance of ECGP. However, the speedup factor and accuracy does increase with problem size, indicating that ECGP may perform substantially better on even larger problems.

The difference in performance between ECGP and CGP on the 3-bit multiplier is due to ECGP exploiting any modularity present in the multiplier problem. We do not believe any undiscovered modular way of building multipliers from multipliers was found. Instead, ECGP simply found and reused functions, which can be constructed from the primitive function set, and conform to the conventional way of building multipliers. Three functions were commonly encoded by evolved modules present

TABLE VI
THE MEDIAN NUMBER OF EVALUATIONS (ME), MEDIAN ABSOLUTE DEVIATION (MAD), AND INTERQUARTILE RANGE (IQR) OF CGP AND ECGP FOR THE 1-BIT, 2-BIT, AND 3-BIT DIGITAL ADDER PROBLEMS. THE U VALUE IS FROM THE MANN–WHITNEY SIGNIFICANCE TEST AND IS HIGHLY SIGNIFICANT ($P < 0.001$) WHEN DENOTED BY[‡]

Adder	CGP			ECGP			U
	ME	MAD	IQR	ME	MAD	IQR	
1	5,923	3,054	7,122	8,825	4,396	8,545	1,449
2	132,565	76,228	178,335	124,251	49,926	99,291	1,060
3	1,943,585	996,482	2,174,500	733,909	307,694	599,132	415 [‡]

TABLE VII
THE COMPUTATIONAL EFFORT FIGURES FOR CGP AND ECGP APPLIED TO THE 2-BIT AND 3-BIT DIGITAL MULTIPLIER PROBLEMS

Multiplier	CGP	ECGP	Speed-up
2	33,602	48,642	0.69
3	24,152,005	4,990,082	4.84

in the solutions found: 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 Fig. 5), and the 2-bit \times 1-bit multiplier, which is constructed from two AND Boolean functions. All three of these functions are considered as partial solutions.

For the 2-bit multiplier problem, enzyme GP has a computational effort figure of 136 080 [22]. Comparing the computational effort figure of enzyme GP with those of CGP and ECGP for the 2-bit multiplier shows CGP and ECGP perform 4.05 and 2.80 times faster than enzyme GP, respectively. However, the function set used by enzyme GP is slightly different to the function set used by CGP and ECGP. Enzyme GP only used the functions AND and XOR, which is a subset of the CGP and ECGP function set, and also makes the problem easier to solve, as it only uses functions found in the compact human design. It is also important to note that computational effort depend on the total resources (in terms of the number of nodes allowed in a genotype and the primitive functions used) that have been allowed. This can also make comparisons difficult (see earlier comments on the efficiency of CGP).

IX. DIGITAL COMPARATOR PROBLEM

The digital comparator problem is relatively unheard of in the GP community. In this paper, we are suggesting it as a possible benchmark for testing GP techniques. The digital comparator problem has the interesting property that the number of outputs remains fixed, while the number of inputs can increase with problem difficulty. Most programmers will have come across the “compare” function, found in the majority of programming languages, which compares two numbers to see if the first number is less than, equal to or greater than the second number. The digital comparator problem is based on the same

principle, except the comparison is done using a digital circuit. The n -bit digital comparator circuit has $2n$ inputs, and three outputs, each representing a state: less than, equal to, or greater than. Only one of the three outputs can have a value of 1 at any time, while the remaining have an output of 0. In this paper, we applied CGP and ECGP to the 1-, 2-, and 3-bit comparators. The function set used is the same as we have seen in earlier problems, consisting of the Boolean functions: AND, NAND, OR, and NOR.

A. Results and Discussion

Computational effort figures are calculated for CGP and ECGP applied to the comparator problem, and are shown in Table IX.

Once again, all 50 runs of CGP and ECGP produced 100% successful solutions. From the results in Table IX, it is observed once again that ECGP performs better than CGP for the more difficult problems. Furthermore, the speedup of ECGP also increases with problem difficulty, indicating it could perform even better than CGP on harder problems. However, CGP performs better on the easier 1-bit comparator problem. This coincides with the findings from other problems in this paper, which suggest module acquisition and the reuse of modules, hinders search performance on easy problems. This could be caused by over exploration of code inside the modules. Alternatively, the preference of modules to primitive functions could restrict the search space at various stages of the evolution process, therefore preventing ECGP from finding the solution. Further investigation is required to verify either of these observations.

Once again, the statistics in Table X support the general trend of the computational effort figures. However, for the 2-bit comparator, the figures for the computational effort and the median number of evaluations give conflicting views as to which technique performs best. A possible explanation for this anomaly, could be the underestimation of the computational effort statistic (as described in Section V-C). Despite this discrepancy, the results of the Mann–Whitney test still class the figures for the 3-bit comparator problem as a highly significant result, thereby supporting the results of ECGP on other large problems.

X. SYMBOLIC REGRESSION PROBLEM

Symbolic regression was suggested by Koza in his first book on GP [1]. Since then it has been widely adopted as a bench-

TABLE VIII
THE MEDIAN NUMBER OF EVALUATIONS (ME), MEDIAN ABSOLUTE DEVIATION (MAD), AND INTERQUARTILE RANGE (IQR) OF CGP AND ECGP FOR THE 2-BIT AND 3-BIT DIGITAL MULTIPLIER PROBLEMS. THE U VALUE IS FROM THE MANN-WHITNEY SIGNIFICANCE TEST AND IS HIGHLY SIGNIFICANT ($P < 0.001$) WHEN DENOTED BY[‡]

Multiplier	CGP			ECGP			U
	ME	MAD	IQR	ME	MAD	IQR	
2	6,197	4,130	8,489	7,485	3,304	9,875	1,423
3	4,030,201	2,181,656	6,110,863	854,579	257,354	773,789	272 [‡]

TABLE IX
THE COMPUTATIONAL EFFORT FIGURES FOR CGP AND ECGP APPLIED TO THE 1-BIT, 2-BIT, AND 3-BIT DIGITAL COMPARATOR PROBLEMS

Comparator	CGP	ECGP	Speed-up
1	4,323	4,803	0.90
2	136,002	128,002	1.06
3	1,320,962	647,682	2.04

mark for testing advancements in the GP field [14]. The aim of the symbolic regression problem is to take two sets of points: one set is the inputs, the other set is the outputs, and to find a function which maps each of the points in the input set to each of the points in the output set, within a certain degree of error. Both the input and outputs sets are of an equal size. The function set used is typically comprised of the functions: addition, subtraction, multiplication, and protected division (division by zero returns a result of 1), which we use for the symbolic regression experiments in this paper. Occasionally, you may also see the use of other functions such as: sine, cosine, log, and exponential, when the desired function requires it.

The fitness function used to distinguish between individuals in the population is the absolute error of the function over all the points in the input set. This is the sum of the absolute difference between the calculated output of each individual and the value for the point from the output set, for all points contained in the output set. The criteria for successfully finding a solution is when the absolute error of each point is within 0.01 of the point in the output set. In this paper, we are trying to evolve solutions to the symbolic regression problems $x^6 - 2x^4 + x^2$ and $x^5 - 2x^3 + x$. Both problems are evaluated using a data set of 50 randomly chosen points from the range $[-1, 1]$. CGP and ECGP are allowed to encode one hundred nodes in their respective genotypes. The maximum module sizes allowed for ECGP were three, five, and eight. This was also an experiment to see if a maximum module size of five always performed the best.

A. Results and Discussion

The computational effort figures for CGP and ECGP applied to the two symbolic regression problems tested are shown in Table XI, along with figures for GP with and without ADFs, which were taken from [2].

In all 50 runs, CGP and ECGP found 100% solutions to both polynomials. The computational effort figures in Table IX show that both CGP and ECGP outperform GP with and without ADFs on both symbolic regression problems by a minimum of 15.3 times for the sextic polynomial and 5.5 times for the quintic polynomial.

Comparing the computational effort figures of CGP and ECGP, it can clearly be seen that CGP performs better than ECGP in two out of the three experiments, but ECGP slightly outperforms CGP on the other experiment. This shows that the use of modules in ECGP has the potential of improving the performance of CGP on problems where the reuse of subfunctions is useful. The statistics in Table XII show a similar trend, however, once again there is a contradictory result between the figures for computational effort and the median number of evaluations. For the sextic polynomial problem, CGP performs better than ECGP with any maximum module size, which once again could possibly be attributed to the underestimation of the computational effort formula. The Mann-Whitney test shows that none of the experiments are significant, so it could be possible to conclude that the use of modules in ECGP produces no benefit to performance on this type of problem. This possibly highlights a lack of modularity in symbolic regression problems, thereby removing any advantage modular approaches, such as ECGP, have over nonmodular approaches like CGP. However, it does highlight the problem of selecting a suitable maximum limit for the size of the modules.

From the computational effort results, it can be seen on both occasions where ECGP performed better than CGP, the maximum limit for the size of the modules is different. In the remaining ECGP experiments, the cost of choosing an unsuitable maximum limit for the size of the modules can decrease the performance of the program by up to a factor of two when compared with CGP. From the previous experiments in this paper, if a problem is too simple, CGP tends to find a solution for the problem, while ECGP is still exploring the search space, as the overheads associated with the additional evolutionary operators of ECGP increases the exploration time required. However, when a problem reaches a certain level of complexity, the acquisition, evolution and reuse of modules in ECGP start to outperform CGP on most of the smaller choices for the maximum size limit of the modules.

PDGP has also been applied to the $x^6 - 2x^4 + x^2$ symbolic regression problem. The results from [14] show PDGP has a computational effort of 91 000, which is much better than GP

TABLE X
THE MEDIAN NUMBER OF EVALUATIONS (ME), MEDIAN ABSOLUTE DEVIATION (MAD), AND INTERQUARTILE RANGE (IQR) OF CGP AND ECGP FOR THE 1-BIT, 2-BIT, AND 3-BIT DIGITAL COMPARATOR PROBLEMS. THE U VALUE IS FROM THE MANN-WHITNEY SIGNIFICANCE TEST AND IS HIGHLY SIGNIFICANT ($P < 0.001$) WHEN DENOTED BY[‡]

Comparator	CGP			ECGP			U
	ME	MAD	IQR	ME	MAD	IQR	
1	771	502	998	809	542	960	1,310
2	33,413	9,642	18,694	38,503	15,738	29,670	1,332
3	265,727	109,194	265,087	179,419	71,166	170,337	697 [‡]

TABLE XI
THE COMPUTATIONAL EFFORT FIGURES FOR CGP AND ECGP (WITH A MAXIMUM MODULE SIZE OF 3, 5, AND 8) APPLIED TO THE SYMBOLIC REGRESSION PROBLEMS: $x^6 - 2x^4 + x^2$ AND $x^5 - 2x^3 + x$

Function	CGP	ECGP-3	ECGP-5	ECGP-8	GP without ADFs	GP with ADFs
$x^6 - 2x^4 + x^2$	55,692	54,353	71,077	76,997	1,440,000	1,176,000
$x^5 - 2x^3 + x$	36,708	59,551	35,896	72,075	396,000	1,200,000

TABLE XII
THE MEDIAN NUMBER OF EVALUATIONS (ME), MEDIAN ABSOLUTE DEVIATION (MAD), AND INTERQUARTILE RANGE (IQR) OF CGP AND ECGP (WITH MAXIMUM MODULE SIZES 3, 5, AND 8) FOR THE SYMBOLIC REGRESSION PROBLEMS $x^6 - 2x^4 + x^2$ AND $x^5 - 2x^3 + x$. THE U VALUE IS FROM THE MANN-WHITNEY SIGNIFICANCE TEST, WHERE CGP IS COMPARED WITH ECGP FOR EACH OF THE MAXIMUM MODULE SIZES. ALL VALUES ARE IN TERMS OF THOUSANDS

Problem	CGP			ECGP-3			ECGP-5			ECGP-8			$U - 3$	$U - 5$	$U - 8$
	ME	MAD	IQR	ME	MAD	IQR	ME	MAD	IQR	ME	MAD	IQR			
$x^6 - 2x^4 + x^2$	12.7	10.9	64.1	29.7	25.1	279.4	43.7	41.0	275.5	38.8	37.0	299.4	1.5	1.5	1.5
$x^5 - 2x^3 + x$	32.2	31.0	525.6	25.9	24.4	296.8	38.9	38.4	411.3	167.5	164.4	664.9	1.2	1.3	1.4

with or without ADFs but is still outperformed by CGP and ECGP with any of the three maximum module sizes tested. This result is interesting as it highlights the performance benefit of implicit reuse in the representations of graph-based GP systems such as PDGP, CGP, and ECGP.

The PushGP system has also been applied to the $x^6 - 2x^4 + x^2$ symbolic regression problem, while investigating the effect of various operators for controlling program size [35]. However, in this instance, CGP and ECGP seem to significantly outperform the PushGP system on the $x^6 - 2x^4 + x^2$ symbolic regression problem (by approximately eight times), this suggests that the instruction set used by PushGP, which gave it an unfair advantage over GP on the even parity problem, may not be so well suited to this problem.

XI. LAWNMOWER PROBLEM

The lawnmower problem was first introduced by Koza in his second book [2] to test the effectiveness of ADFs 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 [14].

In the lawnmower problem, the objective is to guide a lawnmower around a grass lawn, which consists of $n \times m$ squares (where n and m are user defined parameters) and mow all the grass. 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 losing efficiency. 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 center 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 $0 \leq x < n$ and $0 \leq y < m$.

TABLE XIII
THE COMPUTATIONAL EFFORT (IN TERMS OF THOUSANDS) AND SPEEDUP FIGURES FOR CGP, ECGP, PDGP,
AND GP FOR THE LAWNMOWER PROBLEM WITH VARIOUS LAWN SIZES

Size	CGP	ECGP	PDGP	GP(No ADFs)	GP(With ADFs)	Speed-up	Speed-up	Speed-up	Speed-up
	(1)	(2)	(3)	(4)	(5)	(1)&(2)	(1)&(3)	(1)&(4)	(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	-	-	-

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 [2].

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 finally, *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. 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. Both CGP and ECGP were allowed to encode 50 nodes in their respective genotypes. ECGP was allowed a maximum module size of five.

A. Results and Discussion

The computational effort figures for CGP and ECGP applied to the lawnmower problem are shown in Table XIII. The computational effort figures for PDGP and GP (with and without ADFs) were taken from [14] and [2], respectively.

For all 50 independent runs of CGP and ECGP applied to the lawnmower problem, 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. However, the results for CGP and ECGP on the lawnmower problem show a strange correlation compared with the results of the other techniques on the lawnmower problem, and the previous results for CGP and ECGP in this paper. Normally, as the problem difficulty increases, the computational effort increases, but the results for CGP and ECGP on the lawnmower problem do not follow this trend. Instead, the computational effort figures of CGP and ECGP oscillate, so a harder problem is sometimes easier to solve. This is shown more clearly in Fig. 9. A possible explanation for this could be that each node in CGP and ECGP is producing a set of instructions, rather than performing a calculation (as in the previous problems in this paper). This allows the first few nodes of a CGP genotype, in conjunction with the implicit reuse found in the CGP representation, to behave like ADFs, as they each produce a block of instructions, which can be reused. In ECGP, this would be similar to having ADFs inside the modules. Therefore, the oscillations in the computational effort figures could be directly related to the usefulness of the first few nodes in a CGP or ECGP genotype and the amount each node is reused.

On examining Fig. 9, a general but noisy trend can be seen for CGP and ECGP, in which computational effort does increase

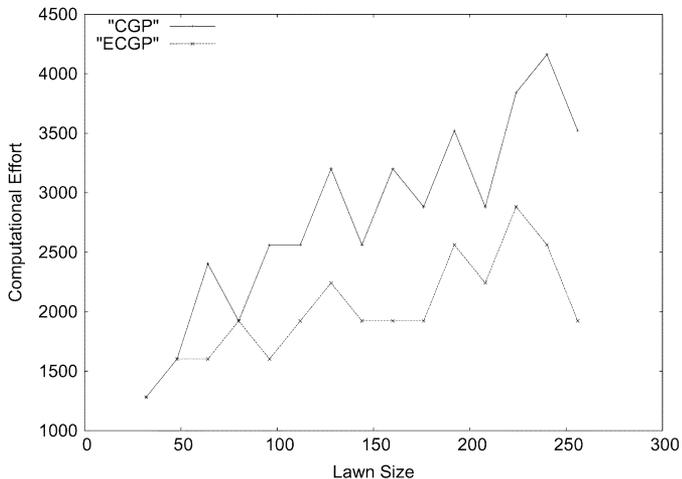


Fig. 9. The unusual correlation between the computational effort figures of CGP and ECGP and increasing lawn size for the lawnmower problem. The general trend of both CGP and ECGP is the computational effort required increases with lawn size.

with problem difficulty. This follows the results of the previous problems discussed in this paper. It can also be seen that the performance speedup of ECGP grows with problem difficulty, suggesting that ECGP could perform even better on larger problems. This speedup can be attributed to the discovery and reuse of subroutines, 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 [23], [25], and [26]. The statistics in Table XIV also support the findings from the computational effort figures and shows that 80% of the experiments are classed as showing some form of significance from the Mann–Whitney test. It could also be said that the significance of the result roughly increases with problem difficulty, as the results that are classed as highly significant are from the hardest problems.

Comparing the computational effort figures for CGP with PDGP (up to a lawn size of 128) and GP without ADFs (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 PDGP and between 14.8 and 1831.4 times faster than GP without ADFs. In fact, CGP even outperforms GP with ADFs on this problem. This result emphasizes the performance gain associated with using a graph based representation (as in CGP and PDGP), rather than a tree-based representation (as in GP). It can also be seen from comparing the two techniques that include a form of ADF (ECGP and GP with ADFs), that ECGP performs between 3.9 and 12.5 times faster than GP with ADFs. Notice also that the speedup grows with the size of the lawn, indicating that ECGP may perform even better than GP with ADFs on larger problems.

Another technique tested on the lawnmower problem is Spector’s GP with ADMs [6]. ADMs are evolved simultaneously with the main GP code in a similar way to ADFs, however, ADMs produce control structures that allow the repeated execution of the ADMs arguments. Spector found that GP with ADMs actually performed worse than GP with ADFs on the 64 square

TABLE XIV
THE MEDIAN NUMBER OF EVALUATIONS (ME), MEDIAN ABSOLUTE DEVIATION (MAD), AND INTERQUARTILE RANGE (IQR) OF CGP AND ECGP FOR THE LAWNMOWER PROBLEM WITH VARIOUS LAWN SIZES. THE U VALUE IS FROM THE MANN–WHITNEY SIGNIFICANCE TEST AND IS marginally significant ($P < 0.05$) WHEN DENOTED BY *, SIGNIFICANT ($P < 0.01$) WHEN DENOTED BY †, AND HIGHLY SIGNIFICANT ($P < 0.001$) WHEN DENOTED BY ‡

Size	CGP			ECGP			U
	ME	MAD	IQR	ME	MAD	IQR	
32	307	124	206	257	96	186	1,046
48	361	150	289	301	94	207	1,011
64	467	200	414	349	130	233	831 †
80	523	182	347	387	142	275	924 *
96	623	204	338	427	126	260	777 ‡
112	581	198	393	493	148	303	1,046
128	711	266	662	561	236	524	922 *
144	659	178	393	493	164	295	833 †
160	639	260	592	475	108	195	828 †
176	625	204	426	529	122	250	901 *
192	781	206	436	579	176	407	916 *
208	717	254	531	511	196	451	935 *
224	751	234	731	643	244	472	955 *
240	855	292	810	561	180	462	688 ‡
256	827	200	446	591	158	330	648 ‡

lawnmower problem, producing a computational effort figure of 18 000. Comparing this result for GP with ADMs with those of CGP and ECGP in Table XIII shows a performance speedup of 7.5 and 11.25 times for CGP and ECGP, respectively.

XII. HIERARCHICAL-IF-AND-ONLY-IF PROBLEM (H-IFF)

The hierarchical if-and-only-if problem (H-IFF) was proposed by Watson *et al.* in the late 1990s, as a more suitable problem for testing the performance of genetic algorithms (GAs) 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 devised to test aspects of the building block hypothesis [36], [37]. It assigns fitness to a bit string according to hierarchies of groups of bits. As you ascend the hierarchy, at each level the number of blocks halve while 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 [38]–[40]), which have been constructed to investigate the role of building blocks in the behavior of GAs. However, H-IFF differs from others by modelling the building block interdependency in a consistent hierarchical fashion. The H-IFF fitness function is deceptive in that it has two possible solutions, a bit

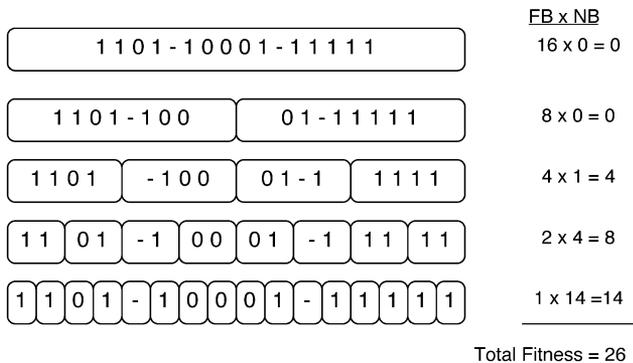


Fig. 10. The H-IFF fitness function for a 16-bit string containing 0’s, 1’s, and blanks (-). FB represents the fitness bonus at each level of the hierarchy for a correct block, and NB represents the number of correct blocks in the current level of the hierarchy.

string containing all 0’s and a bit string containing all 1’s (in a similar way, 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 [41]. Fig. 10 describes the H-IFF fitness function in more detail, and the associated fitness rewards with each level of the hierarchy.

A. Applying CGP to a GA Problem

One of the main issues we faced in attacking this problem was to decide how to apply CGP and ECGP to a problem which is designed for GAs. 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 XI. 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.

For all genotypes, the initial value of all the squares on the tape is blank. This is to remove any bias towards a particular solution (all 0’s or all 1’s). If the tape was initialized at random with blanks, 0’s and 1’s, it is possible the evolved CGP program could be awarded extra fitness points for the 0’s and 1’s on the tape, which are located in squares that were never visited and changed by the tape head (blanks do not contribute towards the fitness score). However, the population of CGP programs is initialized at random, therefore, the first bit-strings produced by the initial population on the tape containing only blanks are random anyway. The starting position for the tape head is in the center of the tape, facing right. In a similar manner 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

$$\begin{aligned}
 &\text{if}(x == \text{blank}), x = 0 \\
 &\quad \text{if}(x == 0), x = 1 \\
 &\quad \text{if}(x == 1), x = 0
 \end{aligned} \tag{2}$$

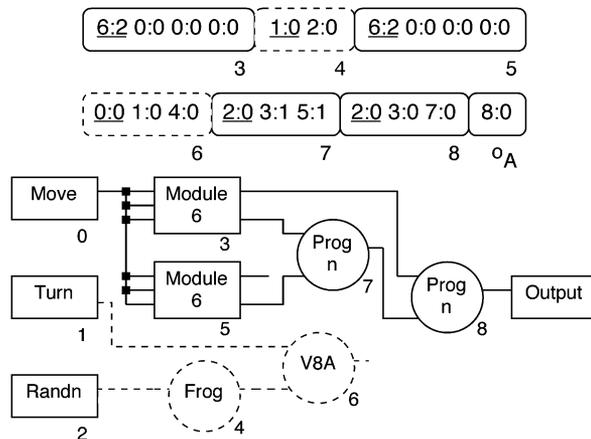


Fig. 11. An ECGP genotype and corresponding phenotype for the 8-bit H-IFF problem. The underlined genes encode the function and node type of each node. The function lookup table is: $v8a(0)$, $frog(1)$, $progn(2)$. See Section XII for details. The index labels are shown underneath each program input 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).

TABLE XV
THE COMPUTATIONAL EFFORT FIGURES FOR CGP AND ECGP (WITH A MAXIMUM MODULE SIZE OF 3, 5, AND 8) APPLIED TO THE H-IFF PROBLEM WITH VARIOUS TAPE LENGTHS

Length	CGP	ECGP-3	ECGP-5	ECGP-8
8	963	1,127	963	1,127
16	2,737	1,926	3,381	2,886
32	3,844	4,329	4,173	4,173
64	5,605	6,726	7,686	7,688
128	5,766	7,042	8,964	8,805
256	8,324	11,526	19,205	15,849

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 (2), *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 $0 \leq r < n$. 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

TABLE XVI

THE MEDIAN NUMBER OF EVALUATIONS (ME), MEDIAN ABSOLUTE DEVIATION (MAD), AND INTERQUARTILE RANGE (IQR) OF CGP AND ECGP (WITH MAXIMUM MODULE SIZES 3, 5, AND 8) FOR THE H-IFF PROBLEM WITH VARIOUS TAPE LENGTHS. THE U VALUE IS FROM THE MANN-WHITNEY SIGNIFICANCE TEST, WHERE CGP IS COMPARED WITH ECGP FOR EACH OF THE MAXIMUM MODULE SIZES. A U VALUE IS marginally significant ($P < 0.05$) IF IT IS DENOTED BY A *, SIGNIFICANT ($P < 0.01$) IF IT IS DENOTED BY A[†], AND HIGHLY SIGNIFICANT ($P < 0.001$) IF IT IS DENOTED BY A[‡]

Length	CGP			ECGP-3			ECGP-5			ECGP-8			$U - 3$	$U - 5$	$U - 8$
	ME	MAD	IQR	ME	MAD	IQR	ME	MAD	IQR	ME	MAD	IQR			
8	125	92	252	155	110	303	147	100	191	159	114	225	1,417	1,388	1,315
16	405	300	1,082	299	196	918	547	400	1,153	423	284	791	1,138	1,314	1,311
32	561	288	1,164	673	462	1,765	599	452	1,859	737	520	2,700	1,369	1,227	1,378
64	891	464	2,463	983	650	1,947	1,109	752	4,626	1,407	1,042	3,868	1,235	2,265 [‡]	1,322
128	861	452	1,615	1,171	664	2,171	1,623	790	1,681	1,415	784	1,887	1,225	1,523	1,397
256	1,473	688	2,190	1,813	954	3,441	3,161	2,036	4,302	2,725	1,838	4,882	1,476	1,660 [†]	1,607 [*]

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 (2). Fig. 11 shows an example of a genotype and corresponding phenotype for the H-IFF problem. However, in principle, the function set chosen for the H-IFF problem may not be the best, as other functions (such as subtraction, multiplication, and if-else statements) could be more useful. The use of other possible functions will be investigated in future investigations. Both CGP and ECGP were allowed to encode 50 nodes in their respective genotypes. Also, ECGP was allowed the maximum module sizes of three, five, and eight.

B. Results and Discussion

The computational effort figures for the results of CGP and ECGP applied to the H-IFF problem are shown in Table XV.

In all 50 independent runs, both CGP and ECGP produced 100% successful solutions. 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, suggesting that CGP may perform better than ECGP on even longer bit-strings. The statistics in Table XVI also support these findings and show that only three results show any form of significance. This tended to be on the problems with a larger tape length where one of the larger module sizes was used. In all cases it showed that the use of modules was detrimental to performance. The performance difference between ECGP and CGP could be attributed to the overhead of module acquisition, evolution and reuse 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 XV 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.* [41], which states the results of a GA with two-point crossover on the 64-bit H-IFF problem. The GA with two-point crossover applied to the 64-bit H-IFF problem, only reached a fitness of 352 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 (1). 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, then $(i + 1) = 400$. If we also give the computational effort calculation the best possible value for $R(z) = 1$, then we can calculate the computational effort of the GA, as shown in (3), with $M = 990$, as the effective population size is 1000-1% elitism (ten individuals)

$$CE = 990 \times 1 \times 400 = 396\,000 \quad (3)$$

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. The results of CGP and ECGP are contrary to the views expressed by Watson *et al.* [41]. Watson *et al.* suggest 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 reuse 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 favorable results are related to the beneficial properties of the genotype-phenotype mapping used in CGP and ECGP, particularly the use of neutrality.

XIII. CONCLUSION AND FUTURE WORK

In this paper, we have presented an extension to CGP, called ECGP, which allows the automatic acquisition, evolution and reuse of subroutines in the form of modules, which are a form of ADF. We have shown both CGP and ECGP perform favorably

when compared with existing techniques such as GP, with and without ADFs, PDGP, enzyme GP, and a GA, on a variety of problems (digital circuits, symbolic regression, lawnmower, and H-IFF). On the larger, more complex problems, ECGP has been shown to outperform CGP, and the speedup grows with problem difficulty, therefore suggesting ECGP may perform even better on harder problems.

One important point, which has been highlighted from the results, is that the performance of ECGP is greatly affected by the choice of the maximum size of the modules it is allowed to acquire. Also, the correlation between the maximum size of the modules and the performance of ECGP is hard to discern, making it even harder to select an optimal maximum module size. This property will be investigated further in future work.

Currently, only a single-level hierarchy is used in ECGP, as modules are not allowed to exist inside other modules. In our future work, we intend to allow the use of embedded submodules. This could lead to the construction of larger, reusable functions making ECGP more effective, as it may allow harder problems to be solved much more quickly, as has been found when ADFs are used in GP [2] and EP [31]. Our recent research suggests a multilevel module hierarchy in ECGP forms a tree structure, in which bloat can occur. Bloat has been found to mainly occur in the unused areas of the module genotype (in the same way as it does in GP), where the section of the code is unmonitored by the fitness function. These unused areas have been found to contain many nested modules, which are completely unused, and the entire subfunction, encoded by the active part of the module genotype, only represents a primitive function. As a side effect, low fitness modules are kept in the module list for extended periods of time, which is disadvantageous to the evolutionary process. These modules could, therefore, be described as exerting certain “parasitic” qualities. We plan to investigate several ideas we currently have to alleviate the problem of bloat and the “parasitic” modules. We will discuss our findings in future papers.

ACKNOWLEDGMENT

The authors of this paper would like to express their thanks to S. L. Smith, X. Yao, and the anonymous reviewers for all of their help and useful comments.

REFERENCES

- [1] J. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press, 1992.
- [2] J. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: MIT Press, 1994.
- [3] J. Angeline and P. J. Pollack, “Evolutionary module acquisition,” in *Proc. 2nd Annu. Conf. Evol. Programming*, 1993, pp. 154–163, MIT Press.
- [4] A. Dessi, A. Giani, and A. Starita, “An analysis of automatic subroutine discovery in genetic programming,” in *Proc. Genetic Evol. Comput. Conf., GECCO 1999*, 1999, pp. 996–1001.
- [5] J. P. Rosca, “Genetic programming exploratory power and the discovery of functions,” in *Proc. 4th Annu. Conf. Evol. Programming*, San Diego, 1995, pp. 719–736.
- [6] L. Spector, “Simultaneous evolution of programs and their control structures,” in *Advances in Genetic Programming II*. Cambridge, MA: MIT Press, 1996, pp. 137–154.
- [7] T. Yu and C. Clack, “Recursion, lambda abstractions and genetic programming,” in *Proc. 3rd Annu. Conf. Evol. Programming*, 1998, pp. 422–431, Morgan Kaufmann.
- [8] J. F. Miller, “An empirical study of the efficiency of learning Boolean functions using a Cartesian genetic programming approach,” in *Proc. Genetic Evol. Comput. Conf., GECCO 1999*, Orlando, Florida, 1999, pp. 1135–1142.
- [9] J. F. Miller and P. Thomson, “Cartesian genetic programming,” in *Proc. 3rd Eur. Conf. Genetic Programming (EuroGP 2000)*, Edinburgh, 2000, vol. 1802, Lecture Notes in Computer Science, pp. 121–132.
- [10] L. Spector, “Autoconstructive evolution: Push, PushGP, and pushpop,” in *Proc. Genetic Evol. Comput. Conf., GECCO 2001*, San Francisco, CA, 2001, pp. 137–146.
- [11] T. Van Belle and D. Ackley, “Code factoring and the evolution of evolvability,” in *Proc. Genetic Evol. Comput. Conf., GECCO 2001*, San Francisco, CA, 2001, pp. 1383–1390.
- [12] J. R. Woodward, “Modularity in genetic programming,” in *Proc. 5th Eur. Conf. Genetic Programming (EuroGP 2003)*, 2003, vol. 2610, Lecture Notes in Computer Science, pp. 258–267.
- [13] V. Khare, B. Sendhoff, and X. Yao, “Environments conducive to evolution of modularity,” in *PPSN IX: Proc. 9th Int. Conf. Parallel Problem Solving from Nature*, Reykjavik, Iceland, Sep. 9–13, 2006, vol. 4193, Lecture Notes in Computer Science, pp. 603–612.
- [14] R. Poli, “Parallel distributed genetic programming,” in *New Ideas in Optimization*, D. Corne, M. Dorigo, and F. Glover, Eds. New York: McGraw-Hill, 1999, pp. 403–432.
- [15] T. Yu and J. F. Miller, “Neutrality and the evolvability of Boolean function landscape,” in *Proc. 4th Eur. Conf. Genetic Programming (EuroGP 2001)*, 2001, vol. 2038, Lecture Notes in Computer Science, pp. 204–217.
- [16] V. K. Vassilev and J. F. Miller, “The advantages of landscape neutrality in digital circuit evolution,” in *Proc. 3rd Int. Conf. Evolvable Syst. (ICES 2000)*, 2000, vol. 1801, Lecture Notes in Computer Science, pp. 252–263.
- [17] V. K. Vassilev, D. Job, and J. F. Miller, “Towards the automatic design of more efficient digital circuits,” in *Proc. 2nd NASA/DoD Workshop on Evolvable Hardware*. Los Alamitos, CA: IEEE Computer Society, 2001, pp. 151–160.
- [18] J. F. Miller, D. Job, and V. K. Vassilev, “Principles in the evolutionary design of digital circuits—Part I,” *Genetic Programming Evol. Mach.*, vol. 1, no. 1, pp. 8–35, 2000.
- [19] J. F. Miller and S. L. Smith, “Redundancy and computational efficiency in Cartesian genetic programming,” *IEEE Trans. Evol. Comput.*, vol. 10, no. 2, pp. 167–174, Apr. 2006.
- [20] E. G. Lopez, R. Poli, and C. A. Coello Coello, “Reusing code in genetic programming,” in *Proc. 7th Eur. Conf. Genetic Programming (EuroGP 2004)*, Coimbra, Portugal, Apr. 5–7, 2004, vol. 3003, Lecture Notes in Computer Science, pp. 359–368.
- [21] C. A. Coello Coello and A. Hernández Aguirre, “Design of combinational logic circuits through an evolutionary multiobjective optimization approach,” *Artif. Intell. Eng., Design, Anal., Manuf.*, vol. 16, no. 1, pp. 39–53, Jan. 2002.
- [22] M. A. Lones and A. M. Tyrell, “Biomimetic representation with genetic programming enzyme,” *Genetic Programming and Evol. Mach.*, vol. 3, no. 2, pp. 193–217, 2002.
- [23] J. A. Walker and J. F. Miller, “Evolution and acquisition of modules in Cartesian genetic programming,” in *Proc. 7th Eur. Conf. Genetic Programming (EuroGP 2004)*, 2004, vol. 3003, Lecture Notes in Computer Science, pp. 187–197.
- [24] H. Schwefel, “Kybernetische Evolution Als Strategie Der Experimentellen Forschung in Der Stromungstechnik,” M.S. thesis, Tech. Univ., Berlin, Germany, 1965.
- [25] J. A. Walker and J. F. Miller, “Investigating the performance of module acquisition in Cartesian genetic programming,” in *Proc. Genetic Evol. Comput. Conf., GECCO 2005*, 2005, vol. 2, pp. 1649–1656.
- [26] J. A. Walker and J. F. Miller, “Improving the evolvability of digital multipliers using embedded Cartesian genetic programming and product reduction,” in *Proc. Int. Conf. Evolvable Syst. (ICES 2005)*, 2005, vol. 3637, Lecture Notes in Computer Science, pp. 131–142.
- [27] S. Christensen and F. Oppacher, “An analysis of Koza’s computational effort statistic for genetic programming,” in *Proc. 5th Eur. Conf. Genetic Programming (EuroGP 2002)*, 2002, vol. 2278, Lecture Notes in Computer Science, pp. 182–191.
- [28] J. Niehaus and W. Banzhaf, “More on computational effort statistics for genetic programming,” in *Proc. 5th Eur. Conf. Genetic Programming (EuroGP 2003)*, 2003, vol. 2610, Lecture Notes in Computer Science, pp. 164–172.

- [29] H. Mann and D. Whitney, "On a test of whether one of 2 random variables is stochastically larger than the other," *Ann. Math. Statistics*, no. 18, pp. 50–60, 1947.
- [30] W. S. Gosset, "The probable error of a mean," *Biometrika*, no. 6, pp. 1–25, 1908.
- [31] K. Chellapilla, "A preliminary investigation into evolving modular programs without subtree crossover," in *Proc. 3rd Annu. Conf. Genetic Programming*, University of Wisconsin, Madison, July 22–25, 1998, pp. 23–31.
- [32] L. Spector and A. Robinson, "Genetic programming and autoconstructive evolution with the push programming language," *Genetic Programming Evol. Mach.*, vol. 3, no. 1, pp. 7–40, Mar. 2002.
- [33] R. Poli and J. Page, "Solving high-order Boolean parity problems with smooth uniform crossover, sub-machine code GP and demes," *Genetic Programming Evol. Mach.*, vol. 1, no. 1, pp. 37–56, 2000.
- [34] J. Torresen, "Evolving multiplier circuits by training set and training vector partitioning," in *Proc. 5th Int. Conf. Evolvable Syst. (ICES 2003)*, Trondheim, Norway, Mar. 17–20, 2003, vol. 2606, Lecture Notes in Computer Science, pp. 228–237.
- [35] R. Crawford-Marks and L. Spector, "Size control via size fair genetic operators in the PushGP genetic programming system," in *Proc. Genetic Evol. Comput. Conf., GECCO 2002*, San Francisco, CA, 2002, pp. 733–739.
- [36] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: Univ. Michigan Press, 1975.
- [37] D. E. Goldberg, *Genetic Algorithms in Search, Optimisation and Machine Learning*. Reading, MA: Addison-Wesley, 1989.
- [38] S. Forrest and M. Mitchell, "Relative building-block fitness and the building-block hypothesis," in *Foundations of Genetic Algorithms 2*. San Mateo, CA, USA: Morgan Kaufmann, 1993.
- [39] M. Mitchell, S. Forrest, and J. H. Holland, "The royal road for genetic algorithms: Fitness landscapes and GA performance," in *Proc. 1st Eur. Conf. Artificial Life*, Cambridge, MA, 1992, MIT Press.
- [40] J. H. Holland, "Royal road functions," in *Proc. Internet Genetic Algorithms Digest*, 1993, vol. 7, no. 22.
- [41] R. A. Watson, G. S. Hornby, and J. B. Pollack, "Modelling building block interdependency," in *Proc. 5th Int. Conf. Parallel Problem Solving from Nature (PPSN V)*, 1998, vol. 1498, Lecture Notes in Computer Science, pp. 97–108.



James Alfred Walker received the B.Sc. degree in mathematics and computer science and the M.Sc. degree in advanced computer science from the University of Birmingham, Birmingham, U.K., in 2002 and 2003 respectively. Since 2004, he has been working towards the Ph.D. degree in electronics at the University of York, York, U.K., under the supervision of J. Miller, and is currently writing his thesis.

He is the coinventor of Embedded Cartesian Genetic Programming, which he has researched and developed for the last five years. He is also involved with the design and maintenance of the Cartesian Genetic Programming website. His research interests include: evolutionary computation, especially genetic programming, the application of evolutionary computation to real-world problems, and image processing and analysis techniques. He is an author or coauthor of ten publications.

Mr. Walker is a Reviewer for the IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION and *Genetic Programming and Evolvable Machines*.



Julian Francis Miller received the B.Sc. degree in physics from the University of London, London, U.K. in 1980, the Ph.D. degree in mathematics from City University, London, in 1988, and the Postgraduate Certificate in teaching and learning in higher education from the University of Birmingham, Birmingham, U.K., in 2002.

He is currently a Lecturer with the Department of Electronics, University of York, York, U.K. He is an author or coauthor of over 130 publications. His research interests include genetic programming, evolvable hardware, and artificial life.

Dr. Miller is an Associate Editor of the IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, *Genetic Programming and Evolvable Machines*. He is an editorial Board Member of the journals *Evolutionary Computation* and *Unconventional Computing*. He has chaired various conferences and workshops in the fields of genetic programming and evolvable hardware.