A Multi-chromosome Approach to Standard and Embedded Cartesian Genetic Programming

Genetic Programming Track

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

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

Rachel Cavill
rc145@ohm.york.ac.uk

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

ABSTRACT

Embedded Cartesian Genetic Programming (ECGP) is an extension of Cartesian Genetic Programming (CGP) that can automatically acquire, evolve and re-use partial solutions in the form of modules. In this paper, we introduce for the first time a new multi-chromosome approach to CGP and ECGP that allows difficult problems with multiple outputs to be broken down into many smaller, simpler problems with single outputs, whilst still encoding the entire solution in a single genotype. We also propose a multi-chromosome evolutionary strategy which selects the best chromosomes from the entire population to form the new fittest individual, which may not have been present in the population. The multi-chromosome approach to CGP and ECGP is tested on a number of multiple output digital circuits. Computational Effort figures are calculated for each problem and compared against those for CGP and ECGP. The results indicate that the use of multiple chromosomes in both CGP and ECGP provide a significant performance increase on all problems tested.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design, Performance

Keywords

Cartesian Genetic Programming, Embedded Cartesian Genetic Programming, Multi-chromosome, Multi-chromosome Evolutionary Strategy, Module Acquisition, Automatically Defined Functions, Evolution, Digital Circuits

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'06, July 8–12, 2006, Seattle, Washington, USA.
Copyright 2006 ACM 1-59593-186-4/06/0007 ...$5.00.

1. INTRODUCTION

For years researchers have been looking to biology for inspiration in finding new ways to increase the performance of Genetic Programming (GP) [8], so that it might be capable of solving larger, more complex problems. One technique called Cartesian Genetic Programming (CGP), which represents programs as directed graphs (rather than trees as in GP) has been shown to be more computationally efficient than GP on a series of problems [13, 12]. Recently CGP has been developed further to form Embedded CGP (ECGP), which is capable of constructing and evolving modules (similar to Automatically Defined Functions in GP [9]) that are called from the main CGP program. ECGP has been shown to be more computationally efficient and to scale better with problem complexity than CGP on a number of problems [22, 24, 23]. In this paper, we propose a multi-chromosome approach to CGP and ECGP, which allows complex problems with multiple outputs to be broken down into many, smaller problems with single outputs that are co-evolved, whilst representing the entire problem in a single genotype. We also introduce a multi-chromosome evolutionary strategy which behaves similarly to an intelligent crossover operator.

The plan for the paper is as follows: section 2 is an overview of CGP, ECGP and other related work. In section 3 we describe the proposed multi-chromosome approach. The details of our experiments are shown in section 4 followed by the results and comparisons for all of the experiments in section 5. Section 6 gives conclusions and some suggestions for future work.

2. BACKGROUND

2.1 Cartesian Genetic Programming (CGP)

CGP was invented by Miller and Thomson [13] for the purpose of evolving digital circuits and represents a program as a directed graph (that for feed-forward functions is acyclic). Originally CGP 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, gave better results when the number of rows was chosen to be one. In CGP, the genotype is a fixed length representation and consists of a list of integers which encode the function and connections of each node in the directed graph. However, the number of nodes in the program (phenotype) can vary but is bounded, as not all of the nodes encoded in the genotype have to be connected. This allows areas of the

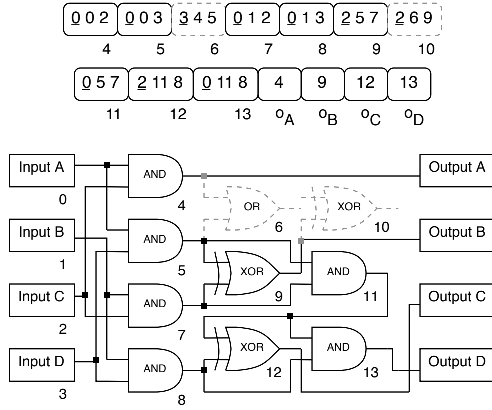

Figure 1: A CGP genotype and corresponding phenotype for the 2-bit multiplier (4 inputs, 4 outputs). The underlined genes in the genotype encode the function of each node, the remaining genes encode the node inputs. The function lookup table is: AND(0), XOR(2), OR(3). The index labels are shown underneath each program input and node in the genotype and phenotype. The inactive areas of the genotype and phenotype are shown in grey dashes.

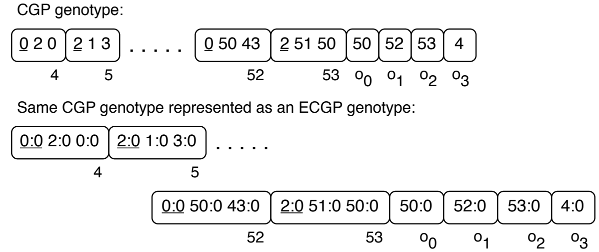

Figure 2: Examples of CGP and ECGP genotypes encoding the same phenotype for the 2-bit multiplier problem (4 inputs, 4 outputs). For each encoded ECGP node, the underlined gene encodes the function and node type, the remaining genes encode the node inputs. Every node encoded in the CGP genotype represents a single output primitive function, therefore every node encoded in the ECGP genotype is of node type 0 (see the text for explanation) and the second integer of each pair encoding the node inputs is always 0. The node index is underneath each node.

genotype to be inactive and have no influence on the phenotype, leading to a neutral effect on genotype fitness called neutrality. This unique type of neutrality has been investigated in detail [13, 25] and found to be extremely beneficial to the evolutionary process on the problems studied.

Each node is encoded by a number of genes. The first gene encodes the function the node represents and the remaining genes encode where 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 the program inputs (terminals). Also, the number of inputs that a node has is dictated by the arity of its function. The program inputs are numbered from 0 to n-1 where n is the number of program inputs. The nodes in the genotype are also numbered sequentially starting from n to n+m-1 where m is the user-determined upper bound of the number of nodes. If the problem requires k program outputs then k integers are added to the end of the genotype, each one encoding a pointer to the output of a node in the graph where the program output is taken from. These k integers are initially set as pointers to the outputs of the last k nodes in the genotype. Fig. 1 shows a CGP genotype and how it is decoded (a 2-bit digital multiplier circuit).

2.2 Embedded Cartesian Genetic Programming (ECGP)

ECGP is currently being developed by Walker and Miller [22] and incorporates ideas from a technique known as Module Acquisition [1] with CGP. This allows for the automatic acquisition, evolution and re-use of partial solutions (referred to as modules) in CGP. This paper only gives a brief overview of ECGP, as there is inadequate space to fully explain the technique. For more information on ECGP, please refer to [22, 24, 23].

The ECGP genotype still consists of a list of integers that encode the connections and functions of each node of the directed graph. However, the ECGP *genotype* is now a variable length representation (in terms of genes) in which the number of genes in the graph is bounded. The number of genes in the ECGP genotype varies as a result of the compression and expansion of modules (how modules are created and destroyed), the re-use of modules elsewhere in the ECGP genotype and the module mutation operators altering the number of module inputs.

Another structural difference is that now each gene is encoded by a pair of integers. For each node encoded in the ECGP genotype, the first integer pair encodes the primitive function or module (by their unique identifier) that the node represents and the node type. Node types allow the identification of nodes encoded in the genotype as follows: primitive functions (node type 0), modules that contain an original section of the genotype (node type I) and modules that contain a re-used section of the genotype (node type II). Different node types need to be identified, as genetic operators act differently on the nodes encoded according to their type. The remaining integer pairs encode the inputs of each node. The first integer of each pair encodes the index of the node or program input (terminal) in the genotype and the second integer encodes the output of the node (nodes in ECGP can have multiple outputs). The number of inputs and outputs that a node has is dictated by the arity of its function. In Fig. 2, an example of how to convert from a CGP to an ECGP genotype is shown.

2.2.1 *The Use of Modules in Embedded Cartesian Genetic Programming*

A module is represented as a bounded variable length genotype that has the same characteristics as an ECGP genotype. The module genotype consists of a list of integers and is split into two parts: the module header and the module body. The module header defines the structure of the module and consists of four integers which encode the module identifier, the number of module inputs, the number of nodes contained in the module and the number of module

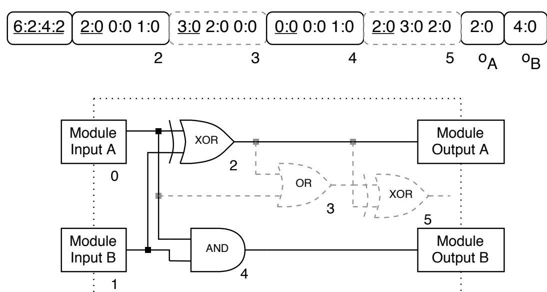

Figure 3: The genotype and corresponding phenotype of a module representing a half adder. The first section of the genotype is the module header. The underlined genes in the module body encode the function of each node, the remaining genes encode the node inputs. The function lookup table is: AND(0), XOR(2), 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.

outputs respectively. The module body defines the connectivity of the module by encoding the connections and functions of the nodes contained in the module and the module outputs (similar to program outputs) in the same manner as an ECGP genotype. An example of a module genotype and corresponding phenotype is shown in Fig. 3.

The size of a module genotype is determined by the number of nodes and module outputs that it encodes. The number of module inputs, nodes contained in the module and the number of module outputs that a module can have is restricted between a lower and upper bound. More details about these restrictions can be found in [22, 24, 23]. In its current form, ECGP only allows modules to contain nodes representing primitive functions rather than nodes representing other modules. The nodes contained inside the module are immune from the genotype point mutation operator, but the module itself is allowed to be mutated by the module mutation operators (including operators for module point mutation, adding/removing module-inputs and adding/removing module-outputs. Further details on the module mutation operators can be found in [22, 24, 23]).

Modules are created by the compress operator by encapsulating all the nodes between two random points in the genotype (in accordance with the module restrictions). Modules can also be destroyed by the expand operator, which reinserts the contents of a module into the genotype, replacing the node representing the module. More information on both of these operators can be found in [22, 24, 23]. Module genotypes are stored in a non-restricted, dynamic module list. The module list is updated every generation with the module list of the fittest individual in the previous generation. This creates a regulatory control of the module list, so that bloat never occurs. The module list is an extension of the primitive function list, in which any node in a genotype could be mutated to to use any available module (allowing the re-use of code elsewhere in the genotype, from different chromosomes in the same genotype) or primitive function (providing that the rules on node type are obeyed. See [22, 24, 23] for more information on node types).

2.3 Multiple Chromosomes in Genetic Programming

There have been several previous approaches which can be said to have used multiple chromosomes within GP [7, 11, 2, 18, 15]. One problem encountered when reviewing the published work on multi-chromosome approaches is the differing definitions of a *chromosome*. Therefore the section starts with publications in which the authors explicitly define as being multi-chromosomal, and then discusses other approaches which may be viewed as being implicitly multi-chromosomal.

Hillis was one of the first to mention multiple chromosomes within the literature in a paper on co-evolving parasites [7]. This paper is mostly cited for its use of co-evolution, but tucked away in the text is a description of how each individual in his system is composed of fifteen pairs of chromosomes. The co-evolved sorting networks which he produces perform well, outperforming the previous human-designed best networks available.

Multiple chromosomes have also been used in other ways within GP. Rick Chow's work on evolving genotype-phenotype mappings [4], described a *data chromosome* and a *mapping chromosome*. The mapping chromosome is a permutation of the numbers 0 to n, where n is the length of the chromosomes. The mapping chromosome is then used to order the data chromosome to produce the final phenotype. The authors' found it effective at solving deceptive problems and some problems which have been identified as GA-hard. Similarly, [5] used an approach which applied a multiploid genotype with a mapping chromosome to the multiple knapsack and set covering problems. However, in this approach the mapping chromosome is used to determine which chromosome each gene in the expressed genotype is taken from.

More recently Mayer investigated how two part crossover, seen in natural systems when multiple chromosomes are used, affected search [11]. The genotype was split up into many chromosomes and a two stage crossover comprising of a standard multi-point crossover and a chromosome shuffling stage was introduced. Chromosome shuffling involves implementing a mixing up of the chromosomes that the child gets from each of the parents, so it may receive chromosomes 1 and 4 from parent 1 and the rest from parent 2. This system was tested on a range of symbolic regression problems and whilst the multi-chromosomal system performed better on some problems, overall there was no clear advantage.

The number of chromosomes which are optimal for a problem has also been investigated [2, 3] and the results on a simple symbolic regression problem seemed to indicate that not only were multiple chromosomes an advantage within the system but having multiple copies of each chromosome also aided evolution.

In addition to those instances where multiple chromosomes have been explicitly used, there are also several systems which could be viewed in a multi-chromosomal way.

Much work has recently been done on the problem of evolving teams of individuals to perform a particular task [10]. Initially work concentrated on traditional team-based problems, like predator-prey problems with multiple predators working together [6]. However, more recently, the problem base has been extended to include symbolic regression

problems and other less obvious problems for teams [18, 19]. When all the members of a heterogenous team are evolved as parts of a single entity, then it seems to be merely a difference in terminology between one that calls these parts the chromosomes in a larger genotype or one that calls them individuals who are part of a larger genotype. Therefore the success of teams across a range of problems, demonstrates the ability of multi-chromosomal systems to be valuable [18, 6, 19].

Another approach which can be viewed within the multi-chromosomal paradigm is that of Multi Expression Programming [15]. Within this system collections of statements are evolved as linear genetic programs. Each statement may call on the previous statements within the genotype to build up more complex expressions, however the fitness of the individual is not predefined to come from the result of any one of the statements. Instead the fitness of every statement is evaluated and the fitness of the individual is taken as being the best of these values. This technique has been used to evolve digital circuits for simple adders and multipliers [14] and the knapsack problem [16].

3. A MULTI-CHROMOSOME APPROACH TO STANDARD AND EMBEDDED CARTESIAN GENETIC PROGRAMMING

3.1 Multi-chromosome Representation

The difference between CGP or ECGP genotype (described earlier in 2.1 and 2.2) and a Multi-chromosome CGP or Multi-chromosome ECGP genotype, is that the Multi-chromosome CGP or Multi-chromosome ECGP genotype is divided into a number of equal length sections called chromosomes. The number of chromosomes present in the genotype of an individual is dictated by the number of program outputs required by the problem, as each chromosome is connected to a *single* program output. This allows large problems with multiple outputs (normally encoded in a single genotype) to be broken down into many smaller problems (each encoded by a chromosome) with a single output. The idea is that this approach should make the problem easier to solve. Already, evolving the outputs to a problem incrementally has been shown to improve evolvability [20], but in this paper we are evolving all of the outputs simultaneously. By allowing each of the smaller problems to be encoded in a chromosome, the whole problem is still encoded in a single genotype but the interconnectivity between the smaller problems (which can cause obstacles in the fitness landscape) has been removed.

Each chromosome contains an equal number of nodes, and is treated as a genotype of an individual with a single program output. The inputs of each node encoded in a chromosome are only allowed to connect to the output of earlier nodes encoded in the same chromosome or any program input (terminals). This creates a form of compartmentalization in the genotype which supports the idea of removing the interconnectivity between the smaller problems encoded in each chromosome. An example of a Multi-chromosome CGP genotype for the 2-bit multiplier problem is shown in Fig. 4. The 2-bit multiplier problem has four outputs, so it is broken down into four smaller problems. Each of the smaller problems has one output and is encoded in a single chromosome.

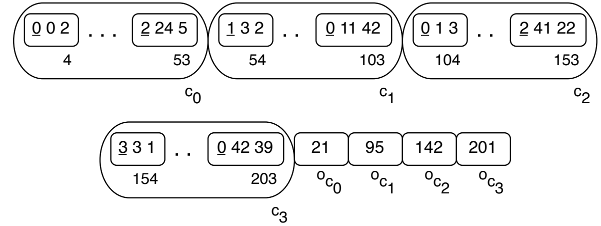

Figure 4: A Multi-chromosome CGP genotype encoding a 2-bit multiplier (four outputs, $o_{c_0} - o_{c_3}$) containing four chromosomes ($c_0 - c_3$), each consisting of fifty nodes

The multi-chromosome approach to CGP and ECGP shares some similarities with another GP technique known as Parisian GP [17], which is inspired by the Michigan Approach to Classifier Systems, in that both techniques form a solution to a problem from sub-solutions. However, in Parisian GP, an individual only represents part of a solution, and the whole solution is made up of a set of individuals from the population. This differs from the multi-chromosome approach to CGP and ECGP, as each chromosome encodes a solution to a distinct sub-problem and the solution to the entire problem is contained in a single individual, which consists of a number of chromosomes, each encoding a different sub-problem. Another difference between the two techniques is that Parisian GP uses two separate fitness functions; a local fitness function to assess each individuals contribution and a global fitness function to evaluate how well the set of individuals solves the problem. Whereas the multi-chromosome approach to CGP and ECGP uses a single fitness function (see Section 3.2) to evaluate how well each chromosome solves the sub-problem it has been assigned. When all of the sub-problems are solved, the entire problem is also solved.

All the chromosomes for an individual are contained in a single genotype, therefore a single module list is used in Multi-chromosome ECGP, which stores modules created in any chromosome contained in the genotype (rather than having an individual module list for each chromosome, which we will investigate in future work). This allows the sharing and re-use of genetic information associated with high fitness, between the chromosomes in the genotype of an individual. Therefore a partial solution which is present in all of the chromosomes, only needs to be constructed in one chromosome, and then re-used in the other chromosomes. In theory, this could re-introduce the interconnectivity between the smaller problems encoded in each chromosome, by exploiting the similarities between the chromosomes. Re-using relevant information from one chromosome in a different chromosome is the same concept as connecting a section of one sub-problem to another sub-problem in the single chromosome approach, but at the same time keeping each of the chromosomes separate.

3.2 Fitness Function and Multi-chromosome Evolutionary Strategy

The fitness function used in multi-chromosome approach is the same as the fitness function used in single chromosome approach except for one small change. The output of each chromosome in multi-chromosome approach is calculated and assigned a fitness value based on the hamming

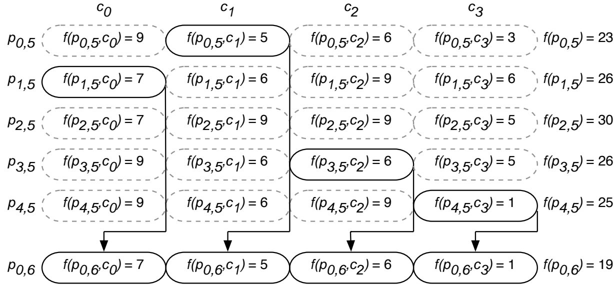

Figure 5: The (1 + 4) multi-chromosome evolutionary strategy used in Multi-chromosome CGP and Multi-chromosome ECGP. $p_{x,g}$ - parent x at generation g, c_y - chromosome y, $f(p_{x,g}, c_y)$ - fitness of chromosome y in parent x at generation g, $f(p_{x,g})$ - fitness of parent x at generation g.

distance from the perfect solution of a single output, whereas in ECGP a fitness values is assigned to the whole genotype based on the hamming distance from the perfect solution over all the outputs (a perfect solution has a fitness of zero). Therefore, the multi-chromosome approach has n fitness values, where n is the number of program outputs, per individual. This allows each chromosome of an individual to be compared with the corresponding chromosome of other individuals in the population by using a (1 + 4) multi-chromosome evolutionary strategy.

The (1 + 4) multi-chromosome evolutionary strategy selects the best chromosome at each position from all of the individuals in the population and generates a new best of generation individual containing the fittest chromosome at each position. The new best of generation individual may not have existed in the population, as it is a combination of the best chromosomes from all the individuals, so it could be thought of as a "super" individual. The multi-chromosome version of the (1 + 4) evolutionary strategy therefore behaves as an intelligent multi-chromosome crossover operator, as it selects the best parts from all the individuals. The overall fitness of the new individual will also be better than or equal to the fitness of any individual in the population from which it was generated. An example of the multi-chromosome evolutionary strategy is shown in Fig. 5.

An outline of the (1 + 4) multi-chromosome evolutionary strategy is shown below:

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

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

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

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

    (a) Select a winning chromosome at each position using the following rules:

        i. If offspring chromosome has a better fitness, the best becomes the winner.

        ii. Otherwise, an offspring chromosome with the same fitness as the best chromosome is randomly selected.

        iii. Otherwise, the parent chromosome remains the winner.

    (b) Go to step a) until all chromosomes are evaluated.

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

4. EXPERIMENT DETAILS

The performance of both the multi-chromosome and single chromosome versions of CGP and ECGP were tested on a number of multiple output problems shown in Table 1. The parameters used for the multi-chromosome and single chromosome versions of CGP and ECGP are shown in Table 2. The probability values chosen for the operators were taken from [24]. The digital multiplier and Arithmetic Logic Unit problems used function set 1, whilst the other problems used function set 2.

5. RESULTS

Computational effort figures were calculated using the formula found in [8] with z=99% and data from fifty independent runs of each problem. The computational effort figures for both the multi-chromosome and single chromosome versions of CGP and ECGP are shown in Table 3.

For all problems tested over fifty runs, CGP, ECGP, Multi-chromosome CGP and Multi-chromosome ECGP produced 100% successful solutions except for the Arithmetic Logic Unit, where CGP and ECGP failed to find a solution after twenty million generations, thus indicating how difficult it is to evolve a solution to the Arithmetic Logic Unit problem. The computational effort figures in Table 3 for CGP and ECGP applied to the Arithmetic Logic Unit are a measure of the computational effort required to find the best possible solution when the runs were stopped after twenty million generations.

Comparing the results for CGP and Multi-chromosome CGP, it is clear to see the use of multiple chromosomes to break down the test problems into smaller, simpler problems provides a distinct advantage. Multi-chromosome CGP significantly outperforms CGP on all of the problems tested. Multi-chromosome CGP improves performance of between approximately, 3 and 392 times when compared with CGP (see Table 3). It is also worth noting that the speedup increases with problem complexity on the adder and multiplier problems, implying that Multi-chromosome CGP may perform even better on larger, more complex problems of this nature. A similar trend is also noticed when comparing the computational effort figures for ECGP and Multi-chromosome ECGP, with a performance increase of between approximately, 1.3 and 80 times. The variance between the speedup times for multi-chromosome CGP and multi-chromosome ECGP (compared with CGP and ECGP) for different problems, appears to be related to the number of problem outputs. Notice how the speedup increases between the 2-bit and 3-bit adder and multiplier problems, as the number of outputs increase on both problems. The biggest speedup recorded was found on the 4x1-bit comparator problem, which is also the problem with the most

Table 1: The experimental problems used to test the performance of the single and multi-chromosome versions of CGP and ECGP.

Problem	Inputs	Outputs	Description
2-bit Adder	5	3	The sum of two 2-bit numbers
3-bit Adder	7	4	The sum of two 3-bit numbers
2-bit Multiplier	4	4	The product of two 2-bit numbers
3-bit Multiplier	6	6	The product of two 3-bit numbers
3:8-bit De-multiplexer	3	8	Uncompresses a compressed 3-bit signal into its eight components
4x1-bit Comparator	4	18	Compares all combinations of inputs for <, =, >
Arithmetic Logic Unit	8	17	Performs addition, subtraction, multiplication & protected division

Table 2: The parameters used for the single chromosome and multi-chromosome versions of CGP and ECGP

Parameter	Value
Population size	5
Initial chromosome size	100 nodes (300 genes)
Initial genotype size	Initial chromosome size x No. of chromosomes
Function set 1	AND, AND (one input inverted), OR, XOR
Function set 2	AND, NAND, OR, NOR
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
Maximum module size	5 nodes
Module list initial state	Empty

Table 3: The computational effort and speedup figures for CGP, ECGP, Multi-chromosome CGP (MC-CGP) and Multi-chromosome ECGP (MC-ECGP)

Problem	CGP (1)	ECGP (2)	MC-CGP (3)	MC-ECGP (4)	Speedup (1) Vs. (3)	Speedup (2) Vs. (4)	Speedup (3) Vs. (4)
2-bit Adder	469,200	311,200	140,800	242,000	3.33	1.29	0.58
3-bit Adder	8,190,400	2,166,000	1,286,000	1,230,400	6.36	1.76	1.05
2-bit Multiplier	52,000	42,000	11,200	22,400	4.64	1.88	0.50
3-bit Multiplier	18,509,600	7,103,600	873,600	867,600	21.19	8.19	1.01
3:8-bit De-multiplexer	75,200	48,400	4,400	6,400	17.09	7.56	0.69
4x1-bit Comparator	3,922,000	1,548,600	10,000	19,200	392.20	80.66	0.52
Arithmetic Logic Unit	100,000,000	100,000,000	1,908,000	1,548,800	52.41	64.57	1.23

outputs. This suggests problem complexity and the number of program outputs are directly linked, implying the multi-chromosome approach is less affected by an increase in problem complexity than the single chromosome approach.

The noticeable speedup caused by the use of multiple chromosomes clearly indicates that by breaking down these complex, difficult problems into smaller, simpler problems, where all interconnections between the smaller problems have been severed, makes the whole problem much easier to solve. The multi-chromosome approach could therefore be used to evolve much harder, multiple-output problems (such as digital circuits), which CGP and ECGP currently fail to solve. The only downside of the multi-chromosome approach is the solutions are much larger (in terms of number of gates used) than the optimal solution (however, our objective in this paper is not to find efficient solutions, our main concern is with improving performance). This appears to be a result of severing the interconnections between the smaller problems, as early sections of the evolved solution which are normally re-used later in the solution are being replicated. A possible approach to reduce the size of the evolved solution is explained in Section 6.

Comparing the results of Multi-chromosome CGP and Multi-chromosome ECGP, shows Multi-chromosome ECGP performs worse than Multi-chromosome CGP on some of the problems where ECGP performs better than CGP. This suggests breaking problems down into smaller problems, reduces the complexity of the problem by such a degree that the overhead of module acquisition in Multi-chromosome ECGP increases the time taken to find a solution. A similar trend is observed for simple problems in ECGP [24, 23]. However, the computational effort figures show Multi-chromosome ECGP does perform better than Multi-chromosome CGP, when a problem reaches a higher level of difficulty (for example the 3-bit multiplier). This suggests the sharing of information between chromosomes is beneficial to the performance of multi-chromosome ECGP. Therefore, even if each sub-problem was encoded in a separate CGP program, and all of the programs were evolved in parallel, multi-chromosome ECGP would still perform better due to the re-use of partial solutions between chromosomes.

To see how much of an impact the multi-chromosome evolutionary strategy had on the results, further experiments were carried out using Multi-chromosome CGP, Multi-chromosome ECGP and a standard $(1 + 4)$ evolutionary strategy on the 2-bit and 3-bit adder problems. The results clearly show the use of a standard $(1 + 4)$ evolutionary strategy with Multi-chromosome CGP and Multi-chromosome ECGP, does not perform as well as Multi-chromosome CGP and Multi-chromosome ECGP with the multi-chromosome evolutionary strategy. This implies the use of the multi-chromosome evolutionary strategy to select the fittest individual in the population (by selecting the fittest chromosomes from each position), is beneficial to the performance of Multi-chromosome CGP and Multi-chromosome ECGP, in contrast to the selection of the fittest individual based on the individual's overall fitness (the sum of all of its chromosome fitness values). However, Multi-chromosome CGP and Multi-chromosome ECGP with a $(1 + 4)$ evolutionary strategy does perform marginally better than the single chromosome versions of CGP or ECGP with a $(1 + 4)$ evolutionary strategy respectively. Therefore implying the multi-chromosome evolutionary strategy is not solely responsible for the improvement in performance, but the use of a multi-chromosome representation, as opposed to a single chromosome representation (as in CGP and ECGP), also improves the performance of both CGP and ECGP.

6. CONCLUSION AND FUTURE WORK

We have presented for the first time a multi-chromosome approach to CGP and ECGP and the use of a multi-chromosome evolutionary strategy. The use of multiple chromosomes in both CGP and ECGP has been shown to significantly speedup performance, when compared with the single chromosome versions of CGP and ECGP on all of the problems tested. This shows that breaking down a large, complex problem into many smaller, simpler problems makes it easier to solve the whole problem. However, in this paper each sub-problem was assigned to a chromosome. In our future work, we intend to allow the number of chromosomes used, and the splitting of the problem into sub-problems, be determined by evolution.

The size of the evolved solutions using multi-chromosome CGP and ECGP, tend to be larger than those found with CGP. However, one advantage of the representation used in the multi-chromosome approach is the whole solution is present in a single genotype, which can be easily converted to a single chromosome genotype (as in CGP or ECGP), by removing the chromosome restrictions. This feature of the representation allows us in our future work to investigate the use of Multi-chromosome CGP and Multi-chromosome ECGP to evolve efficient solutions, using the technique from [21]. Once a solution is found, the fitness function is changed to minimise the total number of nodes used in the phenotype of the solution (either by minimising each chromosome phenotype or the phenotype of the whole solution). This allows the evolved solutions from multi-chromosome CGP and ECGP, to be minimised to a size comparable to those found using CGP or ECGP. Alternatively, the multi-chromosome approach in this paper could also be applied to GP, so that it can be applied to complex, multiple-output problems.

We also intend to investigate the use of neutrality in the Multi-chromosome approach, in which it is possible to have active and inactive *chromosomes* (for example, on a problem with four outputs, twenty chromosomes could be allowed in the genotype, but only four of them would be used simultaneously).

7. REFERENCES

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

[2] R. Cavill, S. L. Smith, and A. M. Tyrrell. Multi-chromosomal Genetic Programming. In *Proc. of the 2005 Genetic and Evolutionary Computation Conference*, volume 2, pages 1649–1656, Washington DC, USA, 25-29 June 2005. ACM Press.

[3] R. Cavill, S. L. Smith, and A. M. Tyrrell. The Performance of Polyploid Evolutionary Algorithms is Improved Both by Having Many Chromosomes and by Having Many Copies of Each Chromosome on Symbolic Regression Problems. In *Proc. of the 2005 Congress on Evolutionary Computation Conference, volume 1*, pages 935–941, sept 2005.

[4] R. Chow. Genotype to Phenotype Mappings with a Multiple-Chromosome Genetic Algorithm. In *Proc. of the 2004 Genetic and Evolutionary Computation Conference*, volume 3102 of *LNCS*, pages 1006–1017, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[5] D. Corne, E. Collingwood, and P. Ross. Investigating Multiploidy's Niche. In *Evolutionary Computing: Selected Papers from the AISB Workshop*, volume 1143 of *LNCS*, pages 189–197, Brighton, UK, Apr. 1996. Springer-Verlag.

[6] T. Haynes, S. Sen, D. Schoenefeld, and R. Wainwright. Evolving a Team. In *Working Notes for the AAAI Symposium on Genetic Programming*, Cambridge, MA, 1995. AAAI.

[7] D. W. Hillis. Co-Evolving Parasites Improve Simulated Evolution in an Optimization Procedure. *Physica D*, 42:228–234, 1990.

[8] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

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

[10] S. Luke and L. Spector. Evolving Teamwork and Coordination with Genetic Programming. In *Genetic Programming 1996: Proc. of the 1st Annual Conference*, pages 150–156, Stanford University, CA, USA, 28–31 1996. MIT Press.

[11] H. A. Mayer and M. Spitzlinger. Multi-chromosomal Representations and Chromosome Shuffling in Evolutionary Algorithms. In *Proc. of the 2003 Congress on Evolutionary Computation Conference*, pages 1145–1149, Canberra, 8-12 Dec. 2003. IEEE Press.

[12] J. F. Miller. An Empirical Study of the Efficiency of Learning Boolean Functions using a Cartesian Genetic Programming Approach. In *Proc. of the 1999 Genetic and Evolutionary Computation Conference*, pages 1135–1142, Orlando, Florida, USA, 1999. Morgan Kaufmann.

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

[14] M. Oltean and C. Grosan. Evolving Digital Circuits using Multi Expression Programming. In *Proc. of the 2004 NASA/DoD Conference on Evolvable Hardware*, pages 87–90, Seattle, 24-26 June 2004. IEEE Press.

[15] M. Oltean, C. Grosan, and M. Oltean. Encoding Multiple Solutions in a Linear Genetic Programming Chromosome. In *International Conference on Computational Science*, pages 1281–1288, 2004.

[16] M. Oltean, C. Grosan, and M. Oltean. Evolving Digital Circuits for the Knapsack Problem. In *Computational Science - ICCS 2004: 4th International Conference, Part III*, volume 3038 of *LNCS*, pages 1257–1264, Krakow, Poland, 6-9 June 2004. Springer-Verlag.

[17] P. Collet, E. Lutton, F. Raynal, M. Schoenauer. Polar IFS + Parisian Genetic Programming = Efficient IFS Inverse Problem Solving. *Genetic Programming and Evolvable Machines*, 1(4):339–361, 2000.

[18] T. Soule. Voting Teams: A Cooperative Approach to Non-typical Problems using Genetic Programming. In *Proc. of the 1999 Genetic and Evolutionary Computation Conference*, pages 916–922, San Francisco, CA, 1999. Morgan Kaufmann.

[19] T. Soule. Heterogeneity and Specialization in Evolving Teams. In *Proc. of the 2000 Genetic and Evolutionary Computation Conference*, pages 778–785, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.

[20] J. Torresen. Evolving Multiplier Circuits by Training Set and Training Vector Partitioning. In *Proc. of the 5th International Conference on Evolvable Systems*, volume 2606 of *LNCS*, pages 228–237, Trondheim, Norway, 17-20Mar. 2003. Springer-Verlag.

[21] V. K. Vassilev, D. Job, and J. F. Miller. Towards the Automatic Design of More Efficient Digital Circuits. In *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*, pages 151–160, Los Alamitos, California, USA, 2001. IEEE Computer Society.

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

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

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

[25] T. Yu and J. F. Miller. Neutrality and the Evolvability of Boolean Function Landscape. In *Proc. of the 4th European Conference on Genetic Programming*, volume 2038 of *LNCS*, pages 204–217, Lake Como, Italy, 18-20 Apr. 2001. Springer-Verlag.