Redundancy and Computational Efficiency in Cartesian Genetic Programming

Julian F. Miller and Stephen L. Smith

***Abstract*—The graph-based Cartesian genetic programming system has an unusual genotype representation with a number of advantageous properties. It has a form of redundancy whose role has received little attention in the published literature. The representation has genes that can be activated or deactivated by mutation operators during evolution. It has been demonstrated that this "junk" has a useful role and is very beneficial in evolutionary search. The results presented demonstrate the role of mutation and genotype length in the evolvability of the representation. It is found that the most evolvable representations occur when the genotype is extremely large and in which over 95% of the genes are inactive.**

***Index Terms*—Cartesian genetic programming (CGP), code bloat, genetic programming, graph-based representations, introns.**

I. Introduction

THERE HAS been considerable interest in the role of redundancy and representation in genetic programming. In this paper, we discuss the properties of the graph-based genetic programming system called Cartesian genetic programming (CGP). This is a representation of programs in which functions are linked by connections, and the genotype is represented in the form of a list of integers. It has an unusual form of redundancy, in which genes may be switched on or off, under the control of evolution. The evolutionary algorithm usually employed in CGP is designed to exploit a process of genetic drift, which has proved to be very effective in a number of studies.

The aim of this paper is to investigate and discuss the characteristics and properties of the genotype representation used in CGP and, in particular, the role of redundancy and its utility in evolutionary search. Results are presented that show unexpectedly high levels of redundancy are notably beneficial to evolutionary search. We also discover empirically that the phenotype length at the end of an evolutionary run has a clear relationship to the genotype length.

II. CGP

A. CGP

CGP [32] is a graph-based form of genetic programming. It was developed from a representation that was used for the evolution of digital circuits [29], [30]. In certain respects, it is similar to the graph-based technique PDGP [42]. In essence, it is characterized by its encoding of a *graph* as a string of integers that represent the functions and connections between graph nodes and program inputs and outputs. This gives it great generality so that it can represent neural networks, programs, circuits, and many other computational structures. Although, in general, it is capable of representing directed multigraphs, it has, so far, only been used to represent directed acyclic graphs. It has a number of features that are distinctive compared with other forms of genetic programming. Foremost among these is that the genotype can encode a nonconnected graph (one in which it is not possible to walk between all pairs of nodes by following directed links). Note that it always encodes at least one connected subgraph. Since disconnected subgraphs can be altered in any way without altering the semantic meaning of the connected graph, there is clearly a many-to-one genotype–phenotype mapping to produce the graph (or program) that is evaluated. The genetic material that is not utilized in the phenotype (that encodes the disconnected subgraphs) is analogous to junk DNA. As we will see, mutations will allow the activation of this redundant code or deactivation of it. Another feature is the ease with which it is able to handle problems involving multiple outputs. Graphs are attractive representations for programs as they are more compact than the more usual tree representation since subgraphs can be used more than once.

CGP has been applied to a growing number of domains and problems: digital circuit design [33], [34], digital filter design [31], image processing [48], [70], artificial life [45], bio-inspired developmental models [36]–[38], evolutionary art [3], and molecular docking [12], [13]. It has been adopted within new evolutionary techniques such as cell-based optimization [44] and social programming [55]. In addition, a more powerful form of CGP with the equivalent of automatically defined functions has recently begun to be developed [56].

In its original formulation, CGP was represented as a directed Cartesian grid of nodes in which nodes were arranged in layers (rows), and it was necessary to specify the number of nodes in each row and the number of columns [29]. The nodes in each column were not allowed to be connected together (rather like a multilayer perceptron neural network). In addition, an additional parameter was introduced called *level-back*, which is defined as how many columns back a node in a particular column could connect to. The program inputs were arranged in an "input layer" on the left of the array of nodes. This is shown in Fig. 1.

It is important to note that in many implementations of CGP, the number of rows (r) is set to one. In this case, the number of columns (c) becomes the *maximum* allowed number of nodes (user defined). Also, the parameter *levels-back* can be chosen to be any integer from one (in which case, nodes can only connect to the previous layer) to the maximum number of nodes (in

Manuscript received November 30, 2004; revised October 5, 2005.

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

Digital Object Identifier 10.1109/TEVC.2006.871253

1089-778X/$20.00 © 2006 IEEE

IEEE PROOF

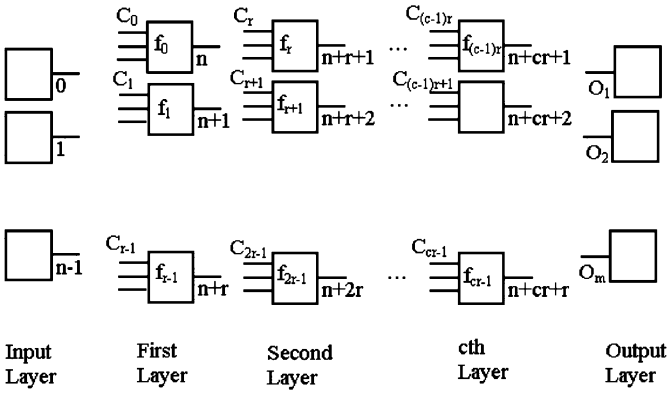

Fig. 1. General form of Cartesian Program for n-input m-output function. There are three user-defined parameters: number of rows (r), number of columns (c), and levels-back. Each node has a set of C_i connection genes (according to arity of function) and single function gene F_i, which defines function of node from a lookup table of available functions. On far left are seen program inputs or terminals, and on far right program output connections O_i.

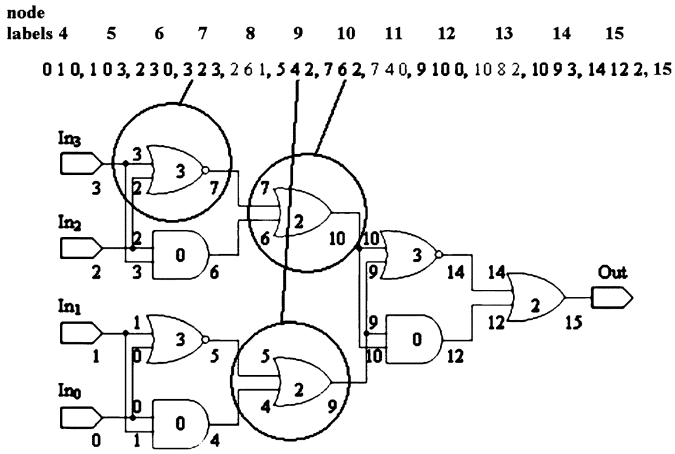

Fig. 2. Cartesian genotype and corresponding phenotype for even four-parity program. Functions f_0-f_3 refer to Boolean operations AND, NAND, OR, and NOR, respectively.

which case a node can connect to *any* previous node). It should also be noted that the output genes can be dispensed with by choosing the program outputs to be taken from the m rightmost consecutive nodes (when only one row is used). In this paper, we report having only used one row and set the levels-back to be equal to the number of columns. We also took the program output from the rightmost node.

The Cartesian genotype is a string of integers

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

where C_i denotes the points that the inputs of the node are connected to. Note that in CGP node 0 described by C_0, f_0 always has an output label that is one more than the number of program inputs (in Fig. 1 shown as n). Each node also has a function f_i chosen from a list of available functions (defined by the user). It also has m output genes O_i, which denote the points where the m program outputs are taken from. Sometimes it happens that the node functions in the function list have different arities. Usually, this is handled by setting the node arity to be the maximum arity that appears in the function list. Nodes with functions that require fewer inputs than the maximum ignore the extra inputs.

If the graphs encoded by the Cartesian genotype are directed, then the range of allowed alleles for C_i are restricted so that nodes can only have their inputs connected to either program inputs or nodes from a previous (left) column. The decoding of the genotype always begins from the rightmost genes. Function values are chosen from the set of available functions. Generally, a single evolutionary operator is used, *point mutation*. In this, a percentage of genes of the number of genes in the genotype are chosen randomly, and alleles are altered to another randomly chosen value (including itself), provided it conforms to the above restrictions.

It should be noted that genotypes are initialized at random so it is possible (though very unlikely) that the program could have only a single node. This could happen if, by chance, the output was taken from a node that was only connected to the program inputs. It is impossible for graphs to have undefined inputs as all node connections are chosen from possible connections on the left and the program inputs are the leftmost points.

In Fig. 2, a genotype is shown as well as how it is decoded (an even four-parity circuit). Although each node must have a function and a set of inputs for that function, the output of a node does not have to be used by the inputs of later nodes. This is shown in Fig. 2, where the outputs of nodes 8, 11, and 13 are not used (shown in gray). These genes are inactive (or temporarily junk) and have a neutral effect on genotype fitness. It should be observed that, although a particular genotype may have a number of such redundant nodes, they cannot be regarded as noncoding genes since, during evolution, mutation may alter genes "downstream" (i.e., on the right) of their position and cause them to be activated and code for something in the phenotype. Similarly, formerly active genes can be deactivated by mutation. This is somewhat akin to the gene switching behavior in real DNA, except that, in that case, genes are switched on and off during the process of translation (i.e., temporally).

Often in the literature, noncoding genes are referred to as introns and coding genes as exons. It is important to note that in CGP introns can exist in the *phenotype*, in addition to the *genotype*. Introns in the phenotype occur when the output of nodes is not affected by changes in connected material (e.g., if the node in question multiplies by zero, an expression defined by connected material). This is the usual form of introns that occurs in tree-based GP. Thus, in CGP, it would be more exact to refer to two types of introns: those that occur in the genotype which might be termed g-introns and those in the phenotype, p-introns. Most GP representations do not have the possibility of g-introns, so the role of this remains neglected. Its role is considered further in Section III. It is well known that genes in real DNA can be active or inactive (and in fact this changes during the reading of DNA in genetic regulatory networks). There are also many sources of redundancy in the natural genetic code. Kimura was one of the first to realize that this redundancy leads to genetic drift in the evolution of biological organisms [16].

IEEE Proof

B. Evolutionary Algorithm

The evolutionary algorithm used in this paper is a form of $1 + \lambda$ evolutionary strategy [47], where $\lambda = 4$, i.e., one parent with four offspring (population size five). Denote the fitness of a genotype g as $f(g)$. Let g_p denote the parent genotype which is mutated λ times to generate λ offspring g_i. Let g'_p be the new parent that is selected from the population $\{g_p, g_i\}$. This is found using the selection procedure

$$g'_p = \begin{cases} g_i, f(g_i) > f(g_j) \forall i \neq j \\ g_i, f(g_i) = f(g_p), \text{choose lowest } i \\ g_p, f(g_p) > f(g_i) \forall i. \end{cases}$$

The second condition in the above algorithm is extremely important. It is used when there are no offspring better than the parent, but there exists at least one that has the same fitness. In this case, the offspring is chosen as the new parent, and it is responsible for genetic drift; generally, the genotype has a large number of inactive genes, and the action of mutation causes many genotypes to decode to exactly the same phenotype and have identical fitness. The constant genetic change in genotypes of the same fitness has been shown to be a main reason for the algorithm's efficiency at solving problems [13], [54], [58], [59].

Although the use of crossover is not ruled out, most implementations of CGP only use point mutation because, as yet, a suitable crossover operator has not been devised. It is very difficult to recombine graphs from two parents in such a way that the offspring inherit characteristics from their parents.

When Cartesian genotypes are initialized, one finds that many of the nodes are inactive. In many CGP implementations, on various problems, it is often found that the number of inactive nodes changes little during evolution. This is supported by our results (see Section III-B). In the next section, we make clear the differences between the genotype–phenotype mapping in CGP and that used in GP. We review published work on bloat in tree-based GP and contrast it with the lack of observed bloat in CGP.

C. Bloat in Tree-Based GP Systems

The problem of the rapid growth of programs produced by genetic programming is very well known and is generally referred to as program bloat [1], [2], [8], [18], [20], [21], [26], [27], [41], [43], [50], [51], [52], [53]. Unfortunately, this growth in program size is almost always due to the growth of pieces of subcode that have little or no semantic effect. Various ideas have been proposed to explain this phenomenon. Originally, it was thought to be caused by *hitchhiking* [53], a process wherby inactive code was propagated by crossover, by virtue of being attached to fitter parents. Another theory was that bloat arose because it provides a protection from the deleterious effects of crossover by increasing the number of crossover points that have no semantic effect on an individual [8], [23], [27], [41]. Another argument put forward was that of *removal bias* [50], [51]. This suggested that there was a natural bias toward large subtree growth because removal of the whole redundant subtree would be disruptive, while enlarging the inoperative inflated code would not change the fitness of the program. Many of the arguments have focused on the crossover operator, though there is no clear reason why these theories might not be similarly applied to mutation operators. There has been work done that suggests that subtree crossover is particularly at fault and mutation to a smaller extent [2].

Much work has focused on the intron view of bloat. Introns are extraneous pieces of code that do not contribute to program fitness. One approach to alleviate the intron problem has been to deliberately insert introns, i.e., insert *explicitly* defined introns [40]. In register machine code GP [4], [39], this can have the effect of automatically suppressing the growth of implicit introns. Recently, however, work has been done that suggests that program growth is not caused by intron growth, but rather intron growth is a *consequence* of program growth. The program growth is linked to the implicit bias in tree-based GP toward deep crossover points because disruption to subtrees near to the program root are likely to be deleterious [26]. This fits with the findings in [28], which showed that throughout the evolutionary run, the nodes closest to the root hardly ever change from those in the initial population. Possibly, the most general argument advanced is that "fitness causes bloat," which asserts that program bloat occurs largely because there are many more larger programs with higher fitness, so the small initial programs drift in this direction [19]–[23].

However, Soule and Heckendorn have found that when crossover adds large branches, it creates offspring with fitness that is independent of the size of the added branch [52]. This does not support the fitness drift theory. Their results support the view that removal bias is the most important factor.

Banzhaf and Langdon presented a very general analysis of the causes of bloat in GP and concluded that if any length unbiased operator is applied that satisfies the relation that the length of the effective (active) code of an individual is solely dependent on the proportion of effective code in the individual, then code growth is inevitable [6]. This result is independent of the fitness landscape, the type of evolutionary operators, and the representation of programs. Their analysis does not apply to CGP because all CGP genotypes are bounded.

The presence of implicit introns in genetic programming is almost universally regarded as bad, yet some researchers have argued that the spread of introns can actually be beneficial in that they provide a natural kind of code compression [6]. It was partly to alleviate the drawbacks of implicit introns that they introduced explicitly defined introns. Other findings have shown that, with genetic algorithms, noncoding regions can actually improve the performance [11], [24], [57]. However, recent studies of tree-based GP systems have been less successful, and it seems that *suppressing* certain types of implicit introns is more beneficial to search [49]. Another study has indicated that introns can improve performance, but only with the imposition of a parsimony function [9].

One of the interesting and advantageous properties of CGP has been the lack of bloat in the phenotype (it cannot occur in the genotype because it is bounded). To date, no work on CGP has required any action to deal with bloat. However, interestingly, a recent development of CGP (called embedded CGP), that allows a variable length genotype, does exhibit bloat [56]. In this paper, results will be presented in Section III that show

IEEE Proof

that bloat is not observed in CGP even when enormous genotypes are allowed. In an earlier investigation, Miller examined this phenomenon in CGP and found it to be intimately connected with the presence of genes that can be activated or deactivated [35]. He argued that when the fitness of genotypes is high, it becomes more likely that *equally good* genotypes will be favorably selected, especially when there are enormous numbers of genotypically different, but phenotypically identical, solutions, as is the case in CGP. In tree-based GP models, most equally good phenotypes differ from one another in useless (bloated) code sections, and they will be strongly selected for when the average population fitness is high. This, unfortunately, propagates the spread of such useless code but paradoxically compresses the useful code [41]. On the other hand, in CGP, the increased proportion of *genetically different but phenotypically identical* code is able to exist without harm (i.e., it does not even have to be processed as it is not in the phenotype).

D. Neutrality and Its Relationship to Evolvability

In many studies it has been argued that genotypic redundancy leading to fitness neutrality can lead to enhanced evolvability [5], [7], [10], [11], [14], [15]. However, recent studies have questioned this. Knowles and Watson examined some of the redundant mappings investigated by Ebner *et al.* [10] and concluded that their supposed evolvability could be produced without redundancy and higher mutation probabilities [17]. Other recent studies have focused on the utility of redundant genotype–phenotype mappings that possess local causality. Rothlauf and Goldberg define a synonymous redundant genotype–phenotype mapping as one in which small gene changes lead to small changes in the phenotype and, consequently, a small change in fitness [46]. They argue that for nondeceptive problems synonymous redundant encodings should be preferred. However, on deceptive problems, nonsynonymous redundant mappings in which small gene changes might lead to large phenotypic changes should perform better. This is because the increased randomness in the search allows a greater ability to escape from suboptima. The genotype–phenotype mapping used in CGP is clearly nonsynonymous according to this definition, as a single gene change could deactivate one phenotype and activate a completely different phenotype that had previously been dormant in the genotype. Our results concur with Rothlauf and Goldberg's observations, as we show that high-computational efficiency can be obtained using the genotype–phenotype mapping employed in CGP (see also [30]). We feel that the discrepancy in our findings, compared with the mappings studied in [10] and those considered by Watson and Knowles [17], relates to the *type* of redundancy that is present in CGP since, in the mappings they studied, they were representations in which all the genes code for phenotypic traits (as opposed to inactive genes in CGP that are simply not read in the genotype-to-phenotype mapping process). In fact, the evidence we assemble later in this paper shows that the genotype–phenotype mapping employed in CGP is at its most effective with extraordinary levels of redundancy (95% of the genes, on average, being redundant). We feel that the unique nature of redundancy in CGP has not been appreciated. However, a full analysis of the reasons that underlie this effectiveness lies outside the compass of this paper and will form a later study.

TABLE I
COMPUTATIONAL EFFORT FOR VARIOUS MUTATION PROBABILITIES AND GENOTYPE SIZES FOR EVEN-THREE PARITY PROBLEM USING GATES AND, OR, NAND, NOR (EACH DATA ENTRY CORRESPONDS TO 100 EVOLUTIONARY RUNS)

GENOTYPE LENGTH	MUTATION PROBABILITY						
	0.01	0.02	0.03	0.04	0.05	0.06	0.08
50	115,204	39,052	25,224	28,208	24,008	12,904	19,504
100	48,008	23,604	18,008	16,404	11,604	13,604	15,404
200	30,020	14,408	12,808	13,608	12,808	7,804	13,004
300	23,208	14,004	9,612	8,808	9,004	12,408	9,604
400	20,408	10,004	8,004	9,612	10,004	8,604	9,004
500	13,804	11,412	7,216	9,608	8,008	10,812	10,404
600	12,612	9,728	7,752	8,016	8,164	7,928	12,848
800	12,628	7,688	5,764	5,904	7,024	7,564	14,232
1000	14,412	7,824	5,604	4,804	6,904	7,564	15,128
2000	7,040	6,816	5,008	7,208	6,664	11,216	12,624
3000	8,852	5,784	6,852	5,304	5,824	7,984	19,092
4000	6,144	3,484	4,644	5,244	7,376	12,208	16,608

III. EXPERIMENTAL INVESTIGATION OF EVOLVABILITY OF CGP WITH RESPECT TO MUTATION PROBABILITIES AND GENOTYPE LENGTHS

A. Experimental Details

The term evolvability is related to how many different phenotypes can be reached by mutating genotypes with the same phenotype. However, it is generally taken to be empirically determined by the number of genotypes that have to be evaluated in order to obtain a particular level of fitness [1]. Experiments have been performed to investigate the relationship between evolvability of Cartesian genotypes and genotype length and mutation probability. Note the genotype lengths set the maximum phenotype (program) lengths. Two problems were considered: even-three parity and the two-bit multiplier. The former Boolean function outputs a one if an even number of its inputs is one; otherwise, it outputs zero. The primitive function set used with even-three parity was {AND, OR, NAND, NOR}. It is well known that evolving a correct even-three parity function is difficult using this function set [18]. The latter is the two-bit multiplier [29], [30], [33], [34]. The two-bit multiplier has four binary inputs and four binary outputs. It calculates the product of two two-bit binary numbers. The primitive functions used for the two-bit multiplier problems was also the set of primitives {AND, OR, NAND, NOR}. To measure the difficulty of evolving these functions, we have used minimum computational effort. This was a measure suggested by Koza [18]. It is the minimum number of genotype evaluations required to give a 0.99 probability of success in an evolutionary run. To calculate it, we carried out 100 evolutionary runs for a range of mutation probabilities and genotype lengths.

B. Results

In Tables I and II, we list the minimum computational efforts found for both problems.

The computational effort figure in Table II, corresponding to a genotype length of 4000 nodes (mutation probability $= 0.01$) was actually calculated at a mutation probability of 0.075. The reason for this is that the figure for 0.01 is much higher than

IEEE Proof

TABLE II
COMPUTATIONAL EFFORT FOR VARIOUS MUTATION PROBABILITIES AND GENOTYPE SIZES FOR TWO-BIT MULTIPLIER PROBLEM USING GATES AND, OR, NAND, NOR (EACH DATA ENTRY CORRESPONDS TO 100 EVOLUTIONARY RUNS). FIGURE MARKED * WAS OBTAINED AT MUTATION PROBABILITY OF 0.075. FIGURE AT MUTATION PROBABILITY OF 0.01 WAS 39 376

Genotype length	Mutation probability						
	0.01	0.02	0.03	0.04	0.05	0.06	0.08
50	-	114,268	80,656	78,724	97,204	92,172	163,688
60	234,744	82,820	71,324	73,452	73,948	87,844	118,324
70	103,704	75,628	69,128	63,620	57,364	81,620	168,976
80	103,684	70,588	65,296	57,616	79,688	79,212	143,528
90	146,904	71,072	51,620	59,048	72,048	95,056	136,328
100	77,308	61,944	61,220	60,496	76,816	89,044	154,088
200	69,152	40,808	43,212	63,376	91,220	101,532	227,536
300	57,624	446,52	37,924	59,048	86,412	129,616	290,896
400	37,452	44,176	53,292	76,808	83,284	161,288	373,456
500	42,256	33,132	31,204	66,252	112,324	189,372	506,928
600	38,888	36,724	48,252	78,736	151,228	230,888	473,788
700	39,376	35,048	33,844	87,132	145,452	211,220	811,468
800	34,568	36,488	52,324	100,564	151,212	237,640	823,228
1000	34,576	29,532	56168	96008	179,536	307,220	744,268
2000	29,288	37,204	80,404	182,420	297,616	691,560	-
3000	22,568	44,648	89,044	155,528	438,736	-	-
4000	25,448*	49,448	121,692	229,448	633,648	-	-

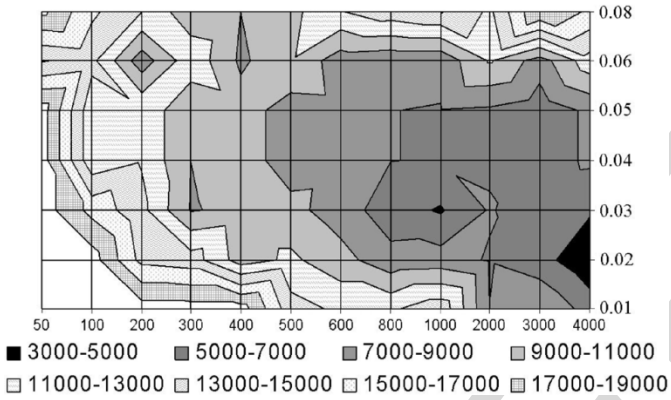

Fig. 3. Computational effort as function of genotype size (x axis, in nodes) versus mutation probability. Lowest computational efforts are seen at high lengths and low mutation probabilities (even-three parity).

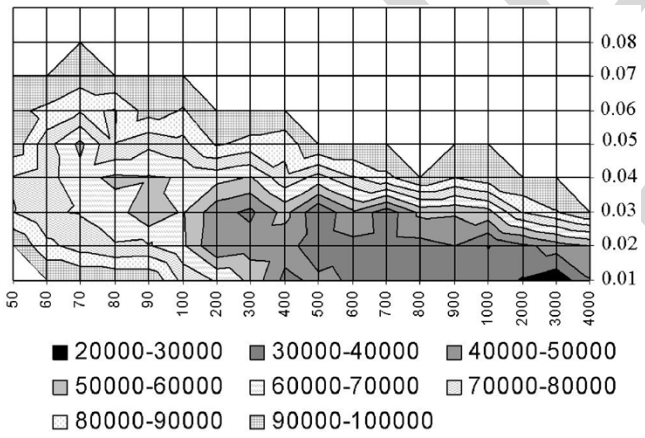

Fig. 4. Computational effort as function of genotype size (x axis, in nodes) versus mutation rate. Lowest computational efforts are seen at high lengths and low mutation rates (two-bit multiplier).

expected, and it was apparent that a lower range of mutation was required to accurately assess minimum computational effort, for such large genotypes. In this paper, we are trying to observe the minimum computational effort across all mutation probabilities, so we thought it fairer to include the term at mutation probability

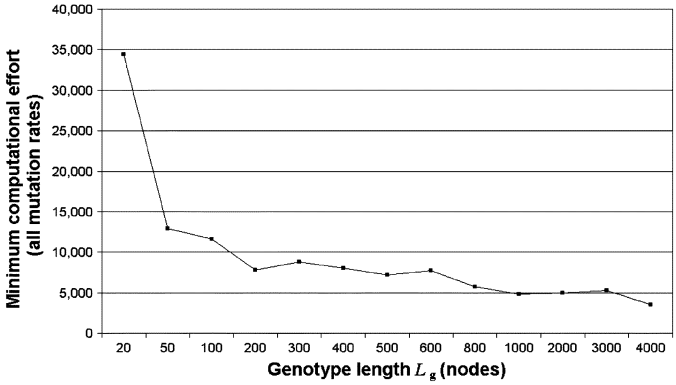

Fig. 5. Minimum computational effort for all mutation probabilities versus genotype length (in nodes) for even-three parity with gate set {AND, OR, NAND, NOR}. Graph shows clearly that computational effort drops consistently with increasing genotype length.

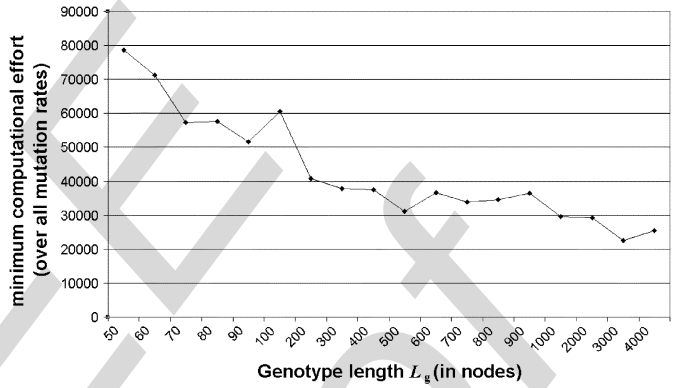

Fig. 6. Minimum computational effort for all mutation probabilities versus genotype length (in nodes) for two-bit multiplier with gate set {AND, OR, NAND, NOR}. Graph shows clearly that computational effort drops consistently with increasing genotype length.

of 0.075. In Figs. 3 and 4, contour plots are given for each table to aid the interpretation of the results. It is very clear from both plots that computational effort is at a minimum for the largest genotypes and small mutation probabilities. It was surprising to the authors that these enormously large genotypes were the most evolvable.

In Figs. 5 and 6, we have also plotted the minimum computational effort for all mutation probabilities for each genotype length. This shows a consistent trend of decreasing effort as the genotype length is increased.

In Fig. 7, we plot the average final phenotype length for all mutation probabilities against the genotype length for the two-bit multiplier problem. We have fitted the data to a polynomial and found that it fits very closely to the curve

$$L_p \cong 2.24 L_g^{0.53}.$$

We have no explanation for this relation and intend to carry out further analysis to discover its origin. We suspect that it may be explained from a mathematical analysis of average phenotype lengths from random samples of the given genotype lengths.

In Table III, we present the average phenotype length for the initial population (randomly generated) and the percentage phenotype length change at the end of the evolutionary run

IEEE Proof

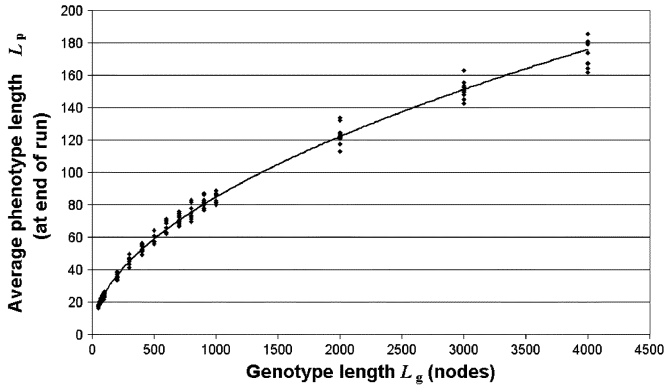

Fig. 7. Average phenotype length at end of an evolutionary run for all mutation probabilities versus genotype length (in nodes) for two-bit multiplier with gate set {AND, OR, NAND, NOR}. Solid line indicates an excellent fit to data and is given by formula above.

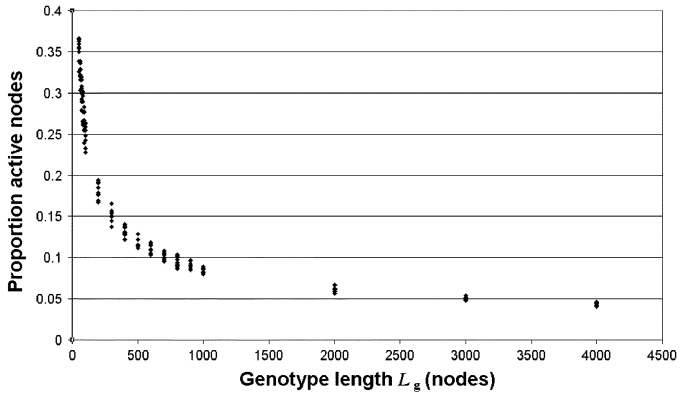

Fig. 8. Average proportion of active nodes in genotype at conclusion of evolutionary run for all mutation probabilities versus genotype length (in nodes) for two-bit multiplier with gate set {AND, OR, NAND, NOR}. Genotypes of length 4000 consist of 5% active nodes and 95% inactive. Note that computational effort is least when most of genotype is inactive.

TABLE III
AVERAGE PHENOTYPE LENGTH OF RANDOMLY GENERATED INITIAL POPULATIONS OF GENOTYPES AND GENOTYPES AT CONCLUSION OF EVOLUTIONARY RUN (WITH 0.01 MUTATION PROBABILITY) FOR TWO-BIT MULTIPLIER WITH GATE SET {AND, OR, NAND, NOR}. FIGURES MARKED * ARE STATISTICALLY SIGNIFICANT

L_g	average L_p of initial population	average increase of L_p (in %) at end of run
50	15.41	18.4*
60	17.67	11.4*
70	19.74	13.1*
80	21.55	10.2*
90	23.55	6.1*
100	25.06	3.2
200	36.84	5.1
300	46.27	7.5
400	54.24	0.8
500	61.51	4.4
600	68.52	3.9
700	72.89	-0.7
800	77.65	6.7
1000	81.89	6.1
2000	86.48	2.6
3000	124.75	7.3
4000	156.96	3.8

of 4000 generations (with mutation probability 0.01) for the two-bit multiplier problem. There is a consistent statistically significant increase in length for modest genotype lengths (up to 90 nodes), but it becomes negligible for greater lengths.

In Fig. 8, we plot the proportion of the genotype the phenotype occupies (at the end of the evolutionary runs, for all mutation probabilities). We see that it is a sharply deceasing function. With genotype lengths of 4000 nodes, the average phenotype is less than 5% of the total! This means that the evolutionary algorithm performs best when almost all the genotype is on average, inactive. This substantially supports findings of other studies of CGP [13], [54], [58], [59] that show that highly neutral search is effective.

Another aspect illustrated by these results is the lack of bloat even when the allowed length of the code is enormous, thus giving plenty of space for phenotypes to expand. We feel this strengthens the view that bloat in other forms of GP is, at least partly, caused by having operators that can change the length of the genotype. In CGP, the genotype is fixed and operators can change the length of the phenotype (i.e., mutation). However, as shown here, this does not, in practice, tend to increase the phenotype length; instead, it remains almost constant.

IV. CONCLUSION

In this paper, we have discussed characteristics of the genotype and phenotype representation in CGP and the unexpected benefits of its redundancy to evolutionary search. The best performance was found to employ extremely high levels of redundancy. Detailed further study is required to ascertain *how* the redundancy is utilized and interacts during evolution to assist problem solving.

ACKNOWLEDGMENT

The authors would like to thank S. Harding for his help with data analysis and preparation. They also gratefully acknowledge the helpful comments of the anonymous referees and Editor F. Rothlauf.

REFERENCES

[1] L. Altenberg, "Emergent phenomena in genetic programming," in *Evolutionary Programming: Proc. 3rd Annu. Conf.*, A. V. Sebald and L. J. Fogel, Eds. Singapore, 1994, pp. 233–241.
[2] P. J. Angeline *et al.*, "Subtree crossover causes bloat," in *Genetic Programming 1998: Proc. 3rd Annu. Conf.*, J. R. Koza *et al.*, Eds. San Mateo, CA: Morgan Kaufmann, 1998, pp. 745–752.
[3] L. Ashmore. (2000) An investigation into Cartesian genetic programming within the field of evolutionary art. Dept. Computer Science, Univ. Birmingham. [Online]. Available: http://www.gaga.demon.co.uk/evoart.htm
[4] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic Programming: An Introduction*. San Mateo, CA: Morgan Kaufmann, 1998.
[5] W. Banzhaf, "Genotype-phenotype-mapping and neutral variation: A case study in genetic programming," in *Proc. Conf. Parallel Problem Solving from Nature III*. New York, 1994, pp. 322–332.
[6] W. Banzhaf and W. B. Langdon, "Some considerations on the reason for bloat," *Genetic Programming Evolvable Machines*, vol. 3, no. 1, pp. 81–91, 2002.
[7] L. Barnett, "Ruggedness and neutrality—The NKp family of fitness landscapes," in *Proc. 6th Int. Conf. Artif. Life*, C. Adami, R. Belew, H. Kitano, and C. Taylor, Eds. Cambridge, MA, 1998, pp. 18–27.

IEEE Proof

[8] T. Blickle and L. Thiele, "Genetic programming and redundancy," in *Genetic Algorithms Within Framework Evolutionary Computation*, J. Hopf, Ed: Max-Planck-Institut für Informatik (MPI-I-94-241), 1994, pp. 33–38. (Workshop KI-94), Saarbrücken).

[9] D. S. Burke, K. A. De Jong, J. J. Grefenstette, C. L. Ramsey, and A. S. Wu, "Putting more genetics into genetic algorithms," *Evol. Comput.*, vol. 6, no. 4, pp. 387–410, 1998.

[10] M. Ebner, P. Langguth, J. Albert, M. Shackleton, and R. Shipman, "On neutral networks and evolvability," *Proc. Congr. Evol. Comput.*, pp. 1–8, 2001.

[11] S. Forrest and M. Mitchell, "Relative building-block fitness and the building-block hypothesis," in *Foundations of Genetic Programming 2*, D. Whitley, Ed. San Mateo, CA: Morgan Kaufmann, 1993.

[12] A. B. Garmendia-Doval, D. S. Morley, and S. Juhos, "Post docking filtering using cartesian genetic programming," in *Proc. 6th Int. Conf. Artificial Evolution*, P. Liardet, Ed., 2003, pp. 435–446.

[13] A. B. Garmendia-Doval and J. F. Miller, "Cartesian genetic programming and the post docking filtering problem," in *Genetic Programming Theory Practice II*, U.-M. O'Reilly, Ed., 2004.

[14] M. A. Huynen, "Exploring phenotype space through neutral evolution," *J. Molecular Evol.*, vol. 43, pp. 165–169, 1996.

[15] M. A. Huynen, P. F. Stadler, and W. Fontana, "Smoothness within ruggedness: The role of neutrality in adaptation," *Proc. Nat. Acad. Sci.*, vol. 93, pp. 397–401, 1996.

[16] M. Kimura, "Evolutionary rate at the molecular level," *Nature*, vol. 217, pp. 624–626, 1968.

[17] J. D. K. Richard and A. Watson *et al.*, "On the utility of redundant encodings in mutation-based evolutionary search," in *Parallel Problem Solving from Nature—Proc. PPSN VII, 7th Int. Conf.*, H.-P. Schwefel *et al.*, Eds.. New York, 2002, pp. 7–11. Lecture Notes in Computer Science.

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

[19] W. B. Langdon, "The evolution of size in variable length representations," in *Proc. IEEE Int. Conf. Evol. Comput.*. Piscataway, NJ, 1998, pp. 633–638.

[20] W. B. Langdon and R. Poli, "Fitness causes bloat: Mutation," in *Proc. EuroGP'98: First European Workshop on Genetic Programming*, W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, Eds. New York, 1998, pp. 37–48.

[21] W. B. Langdon and R. Poli *et al.*, "Why ants are hard," in *GP'98: Proc. 3rd Annual Genetic Programming Conf.*, J. R. Koza *et al.*, Eds.. San Mateo, CA, 1998, pp. 193–201.

[22] W. B. Langdon, T. Soule, R. Poli, and J. Foster, "The evolution of size and shape," in *Advances in Genetic Programming*, L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, Eds. Cambridge, MA: MIT Press, 1999, vol. 3, pp. 163–190.

[23] W. B. Langdon, "Quadratic bloat in genetic programming," in *Proc. Genetic Evol. Comput. Conf.*, D. Whitley, D. Goldberg, E. Cantú-Paz, L. Spector, I. Parmee, and H.-G. Beyer, Eds. San Mateo, CA, 2000, pp. 451–458.

[24] J. R. Levenick, "Inserting introns improves genetic algorithm success rate: Taking a cue from biology," in *Proc. 4th Int. Conf. Genetic Algorithms*, R. K. Belew and L. B. Booker, Eds. San Mateo, CA, 1991, pp. 123–127.

[25] S. Luke, S. Hamahashi, and H. Kitano, "Genetic programming," in *Proc. Genetic Evol. Comput. Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds. San Mateo, CA, 1999.

[26] S. Luke, "Code growth is not caused by introns," *Proc. GECCO-2000: Late Breaking Papers*, pp. 228–235, 2000.

[27] N. F. McPhee and J. D. Miller, "Accurate replication in genetic programming," in *Genetic Algorithms: Proc. 6th Int. Conf.*, L. Eshelman, Ed.. San Mateo, CA, 1995, pp. 303–309.

[28] N. F. McPhee and N. J. Hopper, "Analysis of genetic diversity through population history," in *Proc. Genetic Evol. Comput. Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds. San Mateo, CA, 1999, pp. 1112–1120.

[29] J. F. Miller, P. Thomson, and T. C. Fogarty, "Designing electronic circuits using evolutionary algorithms—Arithmetic circuits: A case study," in *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science: Recent Advancements and Industrial Applications*, D. Quagliarella, J. Periaux, C. Poloni, and G. Winter, Eds. New York: Wiley, 1997, ch. 6.

[30] 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.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds. San Mateo, CA, 1999, pp. 1135–1142.

[31] ——, "Evolution of digital filters using a gate array model," in *Proc. 1st Eur. Workshops Evolutionary Image Analysis, Signal Processing Telecommun.*, vol. 1596, R. Poli *et al.*, Eds.. New York, 1999, pp. 17–30. Lecture Notes in Computer Science, vol. 1596.

[32] J. F. Miller and P. Thomson, "Cartesian genetic programming," in *Proc. 3rd Eur. Conf. Genetic Programming*, vol. 1802, R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, Eds., 2000, pp. 121–132. Lecture Notes in Computer Science.

[33] J. F. Miller, D. Job, and V. K. Vassilev, "Principles in the evolutionary design of digital circuits—Part I," in *J. Genetic Programming Evolvable Machines*, 2000, vol. 1, pp. 8–35.

[34] ——, "Principles in the evolutionary design of digital circuits—Part II," in *J. Genetic Programming Evolvable Machines*, 2000, vol. 1, pp. 259–288.

[35] J. Miller, "What bloat? Cartesian genetic programming on boolean problems," in *Proc. Genetic Evol. Comput. Conf.: Late Breaking Papers*, E. D. Goodman, Ed., 2001, pp. 295–302.

[36] J. F. Miller and P. Thomson *et al.*, "A developmental method for growing graphs and circuits," in *Evolvable Systems: From Biology to Hardware, Proc. 5th Int. Conf. ICES 2003*, vol. 2606, A. M. Tyrrell *et al.*, Eds.. New York, 2003, pp. 93–104. Lecture Notes in Computer Science.

[37] J. F. Miller and W. Banzhaf, "Evolving the program for a cell: From French flags to Boolean circuits," in *On Growth, Form and Computers*, S. J. Kumar and P. J. Bentley, Eds. New York: Academic, 2003, pp. 278–301.

[38] J. F. Miller *et al.*, "Evolving developmental programs for adaptation, morphogenesis, and self-repair," in *Advances in Artificial Life, Proc. ECAL*, W. Banzhaf *et al.*, Eds. New York: Springer-Verlag, 2003, vol. 2801, pp. 256–265. Lecture Notes in Artificial Intelligence.

[39] P. Nordin, "A compiling genetic programming system that directly manipulates the machine code," in *Advances in Genetic Programming*, K. E. Kinnear Jr., Ed. Cambridge, MA: MIT Press, 1994, ch. 14, pp. 311–331.

[40] P. Nordin, F. Francone, and W. Banzhaf, "Explicity defined introns and destructive crossover in genetic programming," in *Advances in Genetic Programming*, P. J. Angeline and K. E. Kinnear Jr., Eds. Cambridge, MA: MIT Press, 1995, vol. 2, pp. 111–134.

[41] P. Nordin and W. Banzhaf, "Complexity compression and evolution," in *Genetic Algorithms: Proc. 6th Int. Conf.*, L. Eshelman, Ed.. San Mateo, CA, 1995, pp. 310–317.

[42] R. Poli, "Evolution of graph-like programs with parallel distributed genetic programming," in *Genetic Algorithms: Proc. 7th Int. Conf.*, T. Bäck, Ed.. San Mateo, CA, 1996, pp. 346–353.

[43] J. Rosca *et al.*, "Generality versus size in genetic programming," in *Genetic Programming 1996: Proc. 1st Annu. Conf.*, J. R. Koza *et al.*, Eds.. Cambridge, MA, 1996, pp. 381–387.

[44] J. Rothermich, F. Wang, and J. F. Miller, "Adaptivity in cell based optimization for information ecosystems," in *Proc. Congr. Evol. Comput.*. Piscataway, NJ, 2003, pp. 490–497.

[45] J. A. Rothermich and J. F. Miller, "Studying the emergence of multicellularity with cartesian genetic programming in artificial life," in *Proc. Genetic Evol. Comput. Conf.: Late Breaking Papers*, E. Cantu-Paz, Ed., 2002, pp. 397–403. AAAI.

[46] F. Rothlauf and D. Goldberg, "Redundant representations in evolutionary computation," *Evol. Comput.*, vol. 11, no. 4, pp. 381–415, 2003.

[47] H. P. Schwefel, "Kybernetische evolution als strategie der experimentelen forschung in der stromungstechnik," M.S. thesis, Technical Univ. Berlin, 1965.

[48] L. Sekanina, *Evolvable Components: From Theory to Hardware Implementations*. New York: Springer Verlag, 2004.

[49] P. W. Smith and K. Harries, "Code growth, explicitly defined introns, and alternative selection schemes," *Evol. Comput.*, vol. 6, no. 4, pp. 339–360, 1998.

[50] T. Soule, J. A. Foster, and J. Dickinson *et al.*, "Code growth in genetic programming," in *Genetic Programming: Proc. 1st Annu. Conf.*, J. R. Koza *et al.*, Eds.. Cambridge, MA, 1996, pp. 215–223.

[51] T. Soule, "Code growth in genetic programming," Ph.D. dissertation, Univ. Idaho, 1998.

[52] T. Soule and R. B. Heckendorn, "An analysis of the causes of code growth in genetic programming," *Genetic Programming Evolvable Machines*, vol. 3, pp. 283–309, 2002.

IEEE Proof

[53] W. A. Tackett, "Recombination, selection, and the genetic construction of computer programs," Ph.D. dissertation, Univ. Southern California, 1994.
[54] V. K. Vassilev and J. F. Miller, "The advantages of landscape neutrality in digital circuit evolution," in *Proc. 3rd Int. Conf. Evolvable Systems: From Biology to Hardware*, vol. 1801, J. F. Miller, A. Thompson, P. Thomson, and T. C. Fogarty, Eds. New York, 2000, pp. 252–263. Lecture Notes in Computer Science.
[55] M. S. Voss, "Social programming using functional swarm optimization," in *Proc. IEEE Swarm Intelligence Symp. (SIS03)*, 2003.
[56] J. A. Walker and J. F. Miller *et al.*, "Evolution and acquisition of modules in Cartesian genetic programming," in *Proc. 7th Eur. Conf. Genetic Programming*, vol. 3003, M. Keijzer *et al.*, Eds.. New York, 2004, pp. 187–197. Lecture Notes in Computer Science.
[57] S. Wu and R. K. Lindsay, "Empirical studies of the genetic algorithm with noncoding segments," *Evol. Comput.*, vol. 3, pp. 121–148, 1995.
[58] T. Yu and J. F. Miller, "Neutrality and evolvability of a boolean function landscape," in *Proc. 4th Eur. Conf. Genetic Programming*, vol. 2038, J. F. Miller, M. Tomassini, and W. Langdon, Eds. New York, 2001, pp. 204–217. Lecture Notes in Computer Science.
[59] T. Yu and J. Miller, "Finding needles in haystacks is not hard with neutrality," in *Proc. Eur. Conf. Genetic Programming*, 2002, pp. 13–25.

Julian F. Miller received the B.Sc. degree in physics from the University of London, London, U.K., in 1980, and the Ph.D. degree in mathematics from City University, in 1988. He received 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 in the Department of Electronics, University of York, Heslington, U.K. His research interests include genetic programming, evolvable hardware, and artificial life. He is an author or coauthor of over 100 publications.

Dr. Miller is an Associate Editor of the journals: 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.

Stephen L. Smith received the B.Sc., M.Sc., and Ph.D. degrees in computer science from the University of Kent, Canterbury, U.K., in 1984, 1986, and 1990, respectively.

He is currently a Senior Lecturer in the Department of Electronics, University of York, Heslington, U.K. His research interests include evolutionary computation with a particular interest in representations and the application of evolutionary algorithms to image processing and medical computing.

Dr. Smith is subject Area Editor for the *Journal of Systems Architecture* and was Programme Chair for the 2005 Conference on Information Processing in Cells and Tissues and organizer of the GECCO Workshop on Medical Applications of Genetic and Evolutionary Computation.

IEEE Proof