# An empirical study of the efficiency of learning boolean functions using a Cartesian Genetic Programming approach

**Julian F. Miller**

School of Computing
Napier University
219 Colinton Road
Edinburgh, EH14 1DJ, UK

## Abstract

A new form of Genetic Programming (GP) called Cartesian Genetic Programming (CGP) is proposed in which programs are represented by linear integer chromosomes in the form of connections and functionalities of a rectangular array of primitive functions. The effectiveness of this approach is investigated for boolean even-parity functions (3,4,5), and the 2-bit multiplier. The minimum number of evaluations required to give a 0.99 probability of evolving a target function is used to measure the efficiency of the new approach. It is found that extremely low populations are most effective. A simple probabilistic hillclimber (PH) is devised which proves to be even more effective. For these boolean functions either method appears to be much more efficient than the GP and Evolutionary Programming (EP) methods reported. The efficacy of the PH suggests that boolean function learning may not be an appropriate problem for testing the effectiveness of GP and EP.

## 1 INTRODUCTION

Since the original development of Genetic Programming (GP) [Koza 92, Koza 94], many different forms have been devised [Banzhaf 98]. Furthermore Evolutionary Programming [Fogel 66] has been developed [Fogel 95] and applied to many of the same problems as GP. This paper looks at one particular problem which has received attention from both camps, namely, the problem of boolean concept learning, and in particular, the even-parity problems. It is well known that the even n-parity functions are extremely difficult to find when searching the space of all n-input boolean functions, if the allowed gates are chosen from the set, {and, or, nand, nor } [Koza 92 ].

In the field of evolvable hardware [Sipper 97] the concept of learning Boolean functions by evolving the connections and functionalities of a network of logic gates has recently been investigated [Iba 96, Miller 97, Miller 98a]. Generally a Genetic Algorithm (GA) is employed and a linear integer chromosome is used to represent the logic network. It is apparent that this form of representation of a digital circuit has a natural generalisation which allows it to be used to solve tasks which are not restricted to binary data. It is this more general form which is referred to as Cartesian Genetic Programming (CGP). It is Cartesian in the sense that the method considers a grid of nodes that are addressed in a Cartesian co-ordinate system. CGP has a number of similarities with Parallel Distributed Genetic Programming (PDGP) [Poli 97] and the graph-based GP system PADO (Parallel Algorithm Discovery and Orchestration) [Teller 95].

In section 2 the basic idea of CGP is explained, and why it naturally allows the development of automatically defined functions (ADFs). In addition it can be used to represent functions of any number of outputs. Also in this section the particular case of CGP which is suitable for boolean concept learning, is described. In section 3 the characteristics of the Genetic Algorithm (GA), and Probabilistic Hillclimber (PH), which are used to evolve logically correct representations of boolean functions, are described. A very large amount of computer processing has been undertaken to obtain results, given in section 4, for the evolution of correct even-3,4,5 parity functions, and the considerably more difficult 2-bit multiplier. The results have been compiled for various population sizes, gate sets, and number of allowed nodes (gates). In some cases the efficiency of the search process as a function of population size, depends strongly on the number of allowed nodes. Comparisons of the efficiency of the GA and PH are given with reported results on even-parity functions for EP and GP. In some cases the GA and PH require about 20 times less evaluations to give a success probability of 0.99.

The primary purpose of this paper is to show that very simple Genetic Algorithms, or, Probabilistic Hillclimbers appear to be much more effective at solving boolean concept learning than much more sophisticated methods employing GP or EP [Koza 94 ], [Chellapilla 98]. Another objective is to show that a GA is sometimes most efficient when a tiny population size is used. It was this fact which suggested to the author that a PH might be more efficient.

## 2 CARTESIAN GENETIC PROGRAMMING

In CGP a program is seen as a rectangular array of nodes. The nodes represent any operation on the data seen at its inputs. Each node may implement any convenient programming construct (if, switch, OR, * etc.). All the inputs whether primary data, node inputs, node outputs, and program outputs are sequentially indexed by integers. The functions of the nodes are also separately sequentially indexed. The chromosome is just a linear string of these integers. The idea is best explained with a simple example. Fig 1. shows the genotype and the corresponding phenotype for a program which implements both the difference in volume between two boxes $V_1$ - $V_2$, and the sum of the volumes, $V_1 + V_2$ , where, $V_1 = X_1X_2X_3$, $V_2 = Y_1Y_2Y_3$. The particular values of the dimensions of the two boxes $X_1$, $X_2$, $X_3$, $Y_1$, $Y_2$, $Y_3$, are labelled 0-5, and are seen on the left. The function set is nominally {0=Plus, 1=minus, 2=multiply, 3=divide, 4=or, 5=xor}, the functions actually used in this example are shown in bold in the genotype and are seen inside the nodes. It is nor necessary for the function types to be embedded in the genotype in this way, they could just as well form a contiguous section of the genome. The program outputs are taken from node outputs 10 and 11, $V_1$ and $V_2$ are each re-used in the calculation of the two outputs.

Genotype

0 1 **2** 3 4 **2** 6 2 **2** 7 5 **2** 8 9 **0** 8 9 **1** 10 11

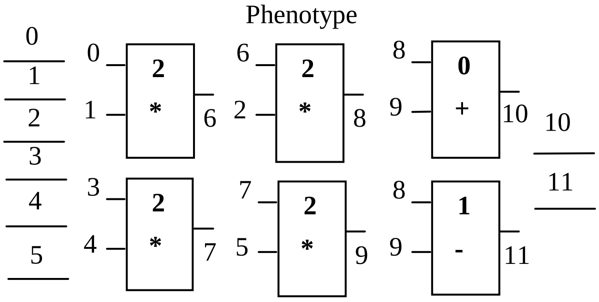

Figure 1: An example CGP genotype and phenotype

If no sequential behaviour is assumed then the inputs of vertical lines of nodes can only be connected to the outputs (or program inputs) which are on the left. The number of columns on the left, which may be connected to a particular cell, is referred to as levels-back. Using a levels-back =1 forces maximum re-use of individual node outputs but hampers large scale re-use of collections of nodes. However using levels-back = number of columns with only a single row allows unrestricted connectivity of nodes and program inputs.

One of the advantages of this representation of a program is that the chromosome representation used is independent of the data type used for the problem, as the chromosome consists of addresses where data is stored. Additionally when the fitness of a chromosome is calculated no interpretation of the genome is required to obtain the addresses in data arrays. Unlike LISP expressions there are no syntactical constraints which must be observed when crossover is carried out. Mutation is very simple one merely has to allow changes to the genes which respect either the functional constraints or the constraints imposed by levels-back. Nodes do not have to be connected and can therefore be redundant, thus the number of nodes used can vary from 0 to the maximum number available. Automatically defined functions emerge quite naturally in this scheme as if a particular collection of gates is very useful then it may be connected many times. In the example shown in Fig 1. There is good re-use of sub-trees with outputs 8 and 9. In the example shown all the nodes have the same number of inputs; this is a convenience, not a fundamental requirement. Thus the representation could be readily generalised to accommodate variable number of inputs and outputs for each node. Whether the representation discussed offers more efficient evolution of programs in general, will have to await further experiments. However the effectiveness of the closely related PDGP [Poli 97] suggests that that signs are favourable.

In this paper a special case of CGP is employed where the data type is binary and the network is allowed to be feed-forward only, this is appropriate for Boolean concept learning. The function set for this is shown in Table 1.

Table 1: Allowed cell functions

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 1 | a | b | $\bar{a}$ | $\bar{b}$ | ab | $a\bar{b}$ | $\bar{a}b$ | $\bar{a}\bar{b}$ |
| 10 | 11 | 12 | 13 | 14 | | | | | |
| $a \oplus b$ | $a \oplus \bar{b}$ | $a + b$ | $a + \bar{b}$ | $\bar{a} + b$ | | | | | |
| 15 | 16 | 17 | 18 | 19 | | | | | |
| $\bar{a} + \bar{b}$ | $a\bar{c} + bc$ | $a\bar{c} + \bar{b}c$ | $\bar{a}\bar{c} + bc$ | $\bar{a}\bar{c} + \bar{b}c$ | | | | | |

All the nodes are assumed to possess three-inputs, if the functions require less, then some connections are ignored, this introduces an additional redundancy into the genome. In Table 1, ab implies a AND b, $\bar{a}$ indicates NOT a, ⊕ represents the exclusive-OR operation and + the OR operation. Functions 0-15 are the basic binary functions of

0, 1 and two inputs. Functions 16-19 are all binary multiplexers with various inputs inverted. The multiplexer (MUX) implements a simple IF-THEN statement (i.e. IF c=0 THEN a ELSE b). These functions (16-19) are called universal logic modules (ULMs). They are well known to be very effective and efficient building blocks for logic circuits [ Chen and Hurst 82].

## 3 CHARACTERISTICS OF THE GENETIC ALGORITHM AND THE PROBABILISTIC HILLCLIMBER

The GA used in this paper is very simple. It is generational in nature, with uniform crossover (50% of genetic material is exchanged), random mutation, and size two probabilistic tournament selection. In this method of parent selection, the fittest chromosome in a tournament is only accepted with a given probability (in this case 0.7), otherwise, the chromosome with the lower fitness is chosen. The amount of genetic recombination is determined by the breeding rate, which represents the percentage of the population, which will take part in recombination. The mutation rate is defined as the percentage of the genes of the entire population, which will undergo mutation. The GA always employs simple elitism where the fittest chromosome of one generation is automatically promoted to the next. There is strong evidence [Miller 98a], that this is extremely beneficial. The fitness of a chromosome is calculated as the ratio of the number of correct output bits divided by the total number of output bits taken over all input combinations. The GA terminates after the chosen number of generations, or when 100% correctness is reached (whichever is the sooner).

The PH algorithm begins with a randomly initialised population of chromosomes. The best chromosome is promoted to the next generation, all the remaining population members are mutations of this chromosome. The process is iterated until termination (same conditions as GA). The only parameters associated with this algorithm are: number of runs, population size, number of generations, and mutation rate. The larger the population the stronger the selection pressure.

The same genotype representation was used for both the GA and PH algorithms.

## 4 DEFINITIONS AND RESULTS

The problems studied in this paper are the even-parity functions, with 3,4, and 5 inputs, and the 2-bit multiplier. The n-bit parity function has n binary inputs, and a single binary output. If the parity is even the output is one if there are an even number of ones in the input stream. The even parity functions of a given number of variables are the most difficult functions to find when carrying out a random search of all GP trees with function set {and, or, nand, nor} [Koza 92]. The n-bit multiplier has 2 n-bit inputs and one 2n-bit output, which is the binary result of multiplying each of the n-bit inputs. It is a difficult function to evolve even when using the complete set of logic gates shown in Table 1. The reason for studying it here is that it differs markedly from the parity functions in that it is built most efficiently with a variety of gates, unlike the parity functions which can be easily built with a single gate (xor).

The method used to assess the effectiveness of an algorithm, or a set of parameters, is that favoured by Koza [Koza 92]. It consists of calculating the number of individual chromosomes, which would have to be processed to give a certain probability of success. To calculate this figure one must first calculate the cumulative probability of success *P(M, i)*, where *M* represents the population size, and *i* the generation number. *R( z)* represents the number of independent runs required for a probability of success (100% functional), given by *z*, by generation *i*. *I(M, z, i)* represents the minimum number of chromosomes which must be processed to give a probability of success *z*, by generation *i*. The formulae for these are given below, $N_s(i)$ represents the number of successful runs at generation *i*, and $N_{total}$ , represents the total number of runs:

$$P(M,i) = \frac{N_s(i)}{N_{total}} \quad , \; R(z) = ceil\left\{\frac{\log(1-z)}{\log(1-P(M,i))}\right\} ,$$

$$I(M, i, z) = \mathrm{M}\, R(z)\, i$$

Note that when $z = 1.0$ the formulae are invalid (all runs successful). In the tables and graphs of this section *z* takes the value 0.99 unless stated otherwise. The variation of *I(M, z, i)* with population size has been investigated for the parity, and multiplier functions. The set of primitive functions used for the parity functions (gate set) was {and, or, nand, nor}, unless stated to the contrary, and for the multiplier all gates were allowed. For the 4-bit even-parity function, *I(M, z, i)* was investigated as a function of M, for three different geometry sizes, 16 x 16, 10 x 10, and 3 x 3, the latter two employed the complete set of allowed primitives (Table 1) Also three geometries were chosen for the 2-bit multiplier, 10 x 10, 7 x 7, and 4 x 4. The different geometries were investigated because the difficulty of the boolean concept learning depends on the amount of resources allocated [Miller 98b], thus it was anticipated that the GA parameters most likely to lead to success would be dependent on this. It should be noted that using *I(M, z, i)* as a measure of computational effort does not directly equate to CPU time when different geometries are being used. A more rigorous treatment would take this into account but the simple object here was to adopt the measure that other researchers have used. It took a great deal of time to collect all the data shown in this section as hundreds of runs of thousands of generations were required for each data point shown in the graphs. In all the tables the figures in parentheses in the *R(z)* column refer to the number of successful runs (out of

100). Thus for instance, in row two of Table 2, *R(z)* is 2, indicating that 2 runs of population 4, lasting 721 generations (including the initial population), is required to give a success probability of 0.99.

## GENETIC ALGORITHM RESULTS

### EVEN PARITY FUNCTIONS

Breeding rate = 100.0%, Crossover rate = 50%, Crossover type = uniform, Mutation rate = 0.25%, acceptance probability = 0.7 (tournament size 2), Unless stated to the contrary gate set = {and, nand, or, nor}. In Tables 4 and 5 all gates were used because with geometries of 10 x 10 and 3 x 3 it was not possible to produce a sufficiently high numbers of 100% functional solutions with the gate set consisting of {and, nand, or, nor}. N denotes the number of generations.

Table 2: 3 bit even parity (geometry = 16 x 16)

| N | Pop. size, M | R(z) | I(M, N, z) |
|---|---|---|---|
| 6,000 | 2 | 1 (100) | 6,002 |
| 4,000 | 4 | 2 (100) | 5,768 |
| 3,000 | 6 | 1 (100) | 4,326 |
| 1,000 | 10 | 2 (100 ) | 5,620 |
| 1,000 | 20 | 1 (100 ) | 6,420 |
| 1,000 | 30 | 1 (100 ) | 7,230 |

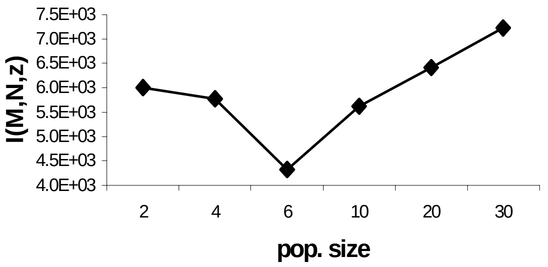

Figure 2: Variation of *I(M,N, z)* with population size for 3-bit even parity (16 x 16)

Table 3: 4 bit even parity (geometry = 16 x 16)

| N | Pop. size, M | R(z) | I(M, N, z) |
|---|---|---|---|
| 10,000 | 4 | 1 (100) | 18,404 |
| 6,000 | 10 | 1 (100 ) | 26,410 |
| 3,000 | 20 | 1 (100 ) | 40,820 |

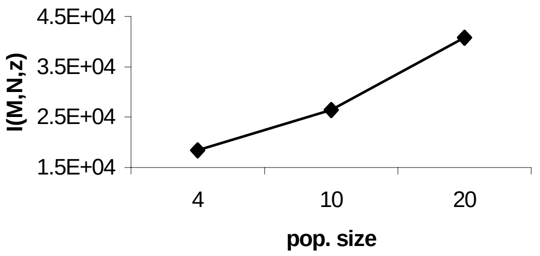

Figure 3: Variation of *I(M,N, z)* with population size for 4-bit even parity (16 x 16)

Table 4: 4 bit even parity

(geometry = 10 x 10, gate set = {all})

| N | Pop. size, M | R(z) | I(M, N, z) |
|---|---|---|---|
| 6,000 | 4 | 1 (100) | 15,364 |
| 6,000 | 10 | 4 (100) | 24,040 |
| 6,000 | 20 | 2 (100) | 19,240 |
| 6,000 | 30 | 3 (100 ) | 21,690 |
| 6,000 | 40 | 2 (100 ) | 28,880 |
| 6,000 | 50 | 1 (100 ) | 30,050 |

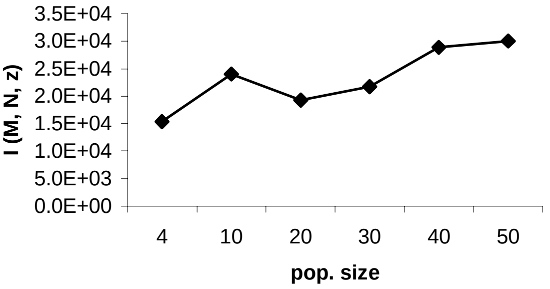

Figure 4: Variation of *I(M,N, z)* with population size for 4-bit even parity (10 x 10, all gates)

Table 5: 4 bit even parity

(geometry = 3 x 3, gate set ={all})

| N | Pop. size, M | R(z) | I(M, N, z) |
|---|---|---|---|
| 25,000 | 10 | 152 (3) | 761,520 |
| 25,000 | 20 | 2 (90) | 110,220 |
| 25,000 | 30 | 1 (100) | 75,150 |
| 10,000 | 40 | 2 (100 ) | 48,240 |
| 5,000 | 50 | 1 (99 ) | 40,400 |

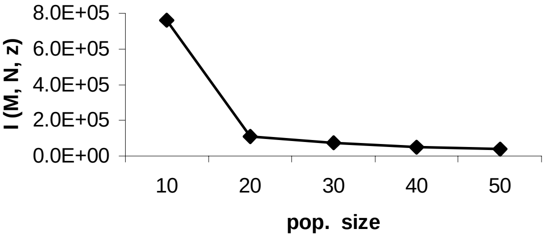

Figure 5: Variation of *I(M,N, z)* with population size for 4-bit even parity (3 x 3, all gates)

Table 6: 5-bit even parity (geometry = 16x16, * = 30x30)

| N | Pop. size, M | R(z) | I(M, N, z) |
|---|---|---|---|
| 15,000 | 4* | 1 (99) | 49,204 |
| 10,000 | 10 | 2 (97 ) | 152,020 |
| 15,000 | 20 | 2 (100 ) | 264,020 |
| 10,000 | 30 | 2 (98 ) | 348,060 |

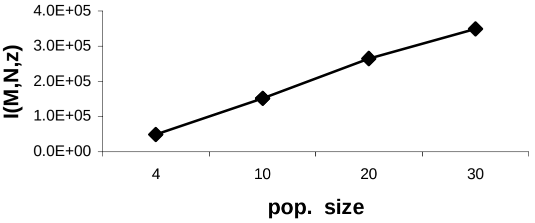

Figure 6: Variation of *I(M,N, z)* with population size for 5-bit even parity ( 16 x 16, population size 4 used a 30 x 30 geometry)

Table 7: 2-bit multiplier (geometry=4x4, gate set={all})

| Pop size | Breeding rate 100% | | Breeding rate 0% | |
|---|---|---|---|---|
| | R(z) | I(M,N,z) | R(z) | I(M,N,z) |
| 6 | 3 (93) | 900,018 | 2 (95) | 816,012 |
| 8 | 2 (93) | 1,344,016 | 2 (95) | 1,312,016 |
| 10 | 4 (88) | 1,760,040 | 3 (88) | 2.040,030 |
| 20 | 3 (97) | 1,560,060 | 2 (95) | 2,080,040 |
| 30 | 2 (99) | 1,800,060 | 3 (97) | 2,340,090 |
| 40 | 3 (96) | 2,880,120 | 3 (99) | 2,400,120 |
| 50 | 2 (97) | 2,600,100 | 2 (100) | 3,200,100 |

Table 8: 2-bit multiplier

(geometry = 7x7, gate set = {all})

| Pop size, | Breeding rate = 100% | | Breeding rate = 0% | |
|---|---|---|---|---|
| | R(z) | I(M,N,z) | R(z) | I(M,N,z) |
| 2 | 1 (99) | 188,002 | 2 (98) | 248,004 |
| 3 | 2 (99) | 264,006 | 1 (100) | 246,003 |
| 4 | 1 (100) | 168,004 | 1 (100) | 192,004 |
| 6 | 1 (100) | 288,006 | 1 (100) | 300,006 |
| 8 | 1 (100) | 256,008 | 2 (100) | 288,016 |
| 10 | 1 (100) | 320,010 | 2 (100) | 360,020 |
| 20 | 1 (100) | 400,020 | 1 (100) | 640,020 |
| 30 | 2 (100) | 480,060 | 1 (100) | 780,030 |
| 40 | 2 (100) | 640,080 | 2 (100) | 1,120,080 |
| 50 | 1 (100) | 800,050 | 1 (100) | 800,050 |

Table 9: 2-bit multiplier

(geometry = 10 x 10, gate set = {all})

| Pop size, | Breeding rate = 100% | | Breeding rate = 0% | |
|---|---|---|---|---|
| | R(z) | I(M,N,z) | R(z) | I(M,N,z) |
| 2 | 1 (100) | 124,002 | 2 (94) | 164,004 |
| 3 | 1 (100) | 156,003 | 2 (98) | 192,006 |
| 4 | 1 (100) | 152,004 | 1 (99) | 160,004 |
| 6 | 1 (100) | 216,012 | 2 (99) | 264,012 |
| 8 | 1 (100) | 176,008 | - | - |
| 10 | 1 (100) | 240,010 | 1 (100) | 300,010 |
| 20 | 1 (100) | 320,020 | 1 (100) | 340,020 |
| 30 | 1 (100) | 360,030 | 1 (100) | 540,030 |
| 40 | 1 (100) | 480,040 | 2 (100) | 640,080 |
| 50 | 2 (100) | 600,100 | 2 (100) | 800,100 |

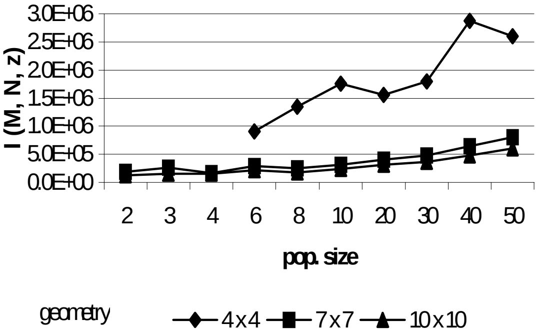

Figure 7: Variation of *I(M,N,z)* with population size, and different geometries (breeding rate=100%)

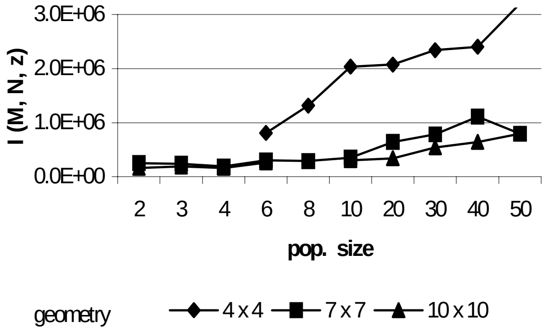

Figure 8: Variation of *I(M,N,z)* with population size, and different geometries (breeding rate=0%)

Figs. 2, 3, 4, 6 demonstrate very clearly that when large numbers of gates are free to be used, the computational effort for correctly evolving the parity functions largely increases with increasing population size. It doesn't seem to depend on the arity. For the 3-bit parity problem the optimum population size seems to be about 6 (Fig. 2). However Fig. 5 shows that when the maximum number of allowed nodes is much smaller, the dependence of computational effort with population size is reversed. Figs. 7 and 8 show the variation of *I(M,N, z)* with population size for the 2-bit multiplier. Three geometries were examined for 100% breeding and 0%. Again the growth of effort with increasing population size is observed. The smaller geometry 4 x 4 doesn't show the inverse dependency with population, which was seen in Fig 5. It may be that 4 x 4 is still large enough for there to be a reasonable density of solutions (the minimum number of gates required to build the multiplier is 7). Comparing Figs. 7 and 8 with each other reveals that the use of recombination reduces computational effort, but only marginally.

## PROBABILISTIC HILLCLIMBER RESULTS

In all the experiments with the probabilistic hillclimber algorithm, the mutation rate per chromosome was set at 1%, thus for the parity functions and an array of 16 x 16 gates the number of genes mutated per chromosome is 10. In the case of the 2-bit multiplier with 10 x 10 geometry, this figure becomes 4. The levels-back parameter was set to 2 for all experiments. For all parity experiments the gate set is {and, or, nand, nor} as in the GA experiments. For the parity functions the geometry was fixed at 16 x 16.

## EVEN PARITY FUNCTIONS

Table 9: 3-bit even parity

| N | Pop. size, M | R(z) | I(M, N, z) |
|---|---|---|---|
| 6,000 | 2 | 1 (100) | 3,122 |
| 5,000 | 3 | 1 (100) | 1,803 |
| 4,000 | 4 | 1 (100) | 2,564 |
| 4,000 | 6 | 1 (100) | 1,926 |
| 6,000 | 8 | 1 (100 ) | 2,888 |
| 6,000 | 10 | 1 (100 ) | 2,410 |
| 6,000 | 20 | 1 (100 ) | 3,220 |
| 6,000 | 30 | 2 (100 ) | 4,860 |
| 1,000 | 40 | 1 (100 ) | 4,040 |
| 1,000 | 50 | 4 (100 ) | 4,200 |

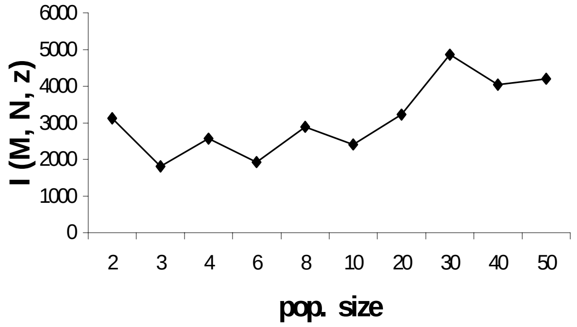

Figure 9: Variation of *I(M,N,z)* with population size for 3-bit even parity function (16 x 16)

Table 10: 4-bit even parity (4,000 generations)

| Pop. size, M | R(z) | I(M, N, z) |
|---|---|---|
| 2 | 3 (100) | 20,346 |
| 3 | 1 (100) | 11,013 |
| 4 | 1 (100) | 9,884 |
| 5 | 1 (100) | 7,005 |
| 6 | 1 (100 ) | 9,726 |
| 8 | 1 (100 ) | 8,328 |
| 10 | 1 (100 ) | 9,010 |
| 20 | 1 (100 ) | 9,220 |
| 30 | 1 (100 ) | 19,830 |
| 40 | 1 (100 ) | 17,640 |
| 50 | 1 (100 ) | 21,550 |

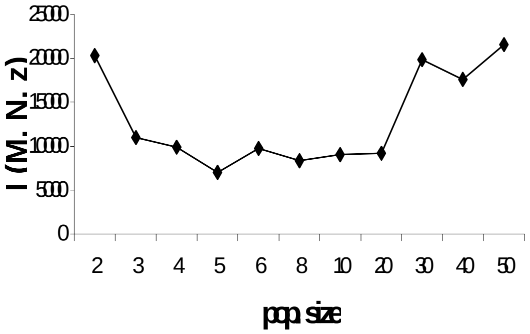

Figure 10: Variation of I(M,N,z) with population size for 4-bit even parity function (16 x 16)

Table 11: 5-bit even parity (10,000 generations)

| Pop. size, M | R(z) | I(M, N, z) |
|---|---|---|
| 2 | 5 (65) | 93,010 |
| 3 | 2 (96) | 49,206 |
| 4 | 2 (98) | 44,008 |
| 5 | 1 (99) | 33,005 |
| 6 | 1 (99 ) | 46,812 |
| 8 | 2 (99 ) | 46,416 |
| 10 | 1 (99 ) | 38,010 |
| 20 | 1 (100 ) | 66,020 |
| 30 | 1 (100 ) | 66,030 |
| 40 | 2 (100 ) | 120,080 |
| 50 | 1 (100 ) | 85,050 |

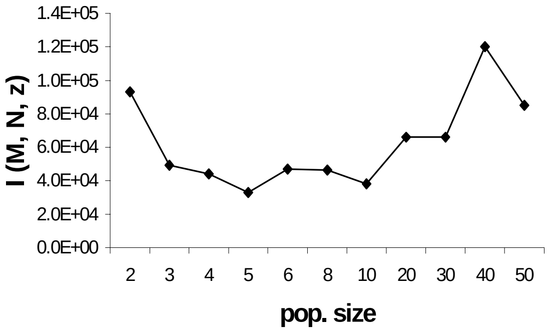

Figure 11: Variation of I(M,N,z) with population size for 5-bit even parity function (16 x 16)

2-BIT MULTIPLIER

For all experiments the geometry = 10 x 10, gate set = {all}, the maximum number of generations was 80,000. *R(z)* = 1 in all these cases. All 100 runs were successful in all cases.

Table 12: 2-bit multiplier

| Pop. size, M | I(M,N,z) |
|---|---|
| 2 | 55,042 |
| 3 | 39,363 |
| 4 | 37,124 |
| 6 | 42,246 |
| 10 | 60,810 |
| 20 | 83,220 |
| 30 | 134,430 |
| 40 | 128,040 |
| 50 | 160,050 |

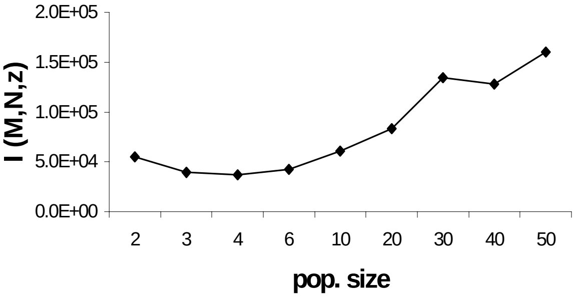

Figure 12: Variation of I(M,N,z) with population size for 2-bit multiplier (10 x 10)

Table 13: Previous published results

| Problem | GP<br>Koza 94 | EP<br>Chellapilla 98 |
|---|---|---|
| 3-bit even parity | 64,000$^{*}$ | 63,000 |
| 4-bit even parity | 176,000$^{*}$ | 118,500$^{*}$ |
| 5-bit even parity | 464,000$^{*}$ | 126,000 |
| 2-bit multiplier | - | - |

Table 14: Results with *z = 1.0*

| Problem | GA | PH |
|---|---|---|
| 3-bit even parity | 7,810 (10) | 1,926 (6) |
| 4-bit even parity | 21,604 (4) | 8,205 (5) |
| 5-bit even parity | 60,004 (4) | 34,010 (20) |
| 2-bit multiplier | 132,002 (2) | 49,924 (4) |

Table 15: Results with *z = 0.99*

| Problem | GA | PH |
|---|---|---|
| 3-bit even parity | 4,326 (6) | 1,803 (3) |
| 4-bit even parity | 18,404 (4) | 7,005 (5) |
| 5-bit even parity | 49,204 (4) | 33,005 (5) |
| 2-bit multiplier | 124,002 (2) | 37,124 (4) |

Table 13 shows some results reported for GP and EP with ADFs (* indicates *z = 1.0*). In Tables 14 and 15 the best results are collated, corresponding to the most favourable population size (shown in parentheses), for success probabilities of *z = 0.99* and *z = 1.0*. It is clear that the computational effort for evolving the functions studied is considerably less for a low population Genetic Algorithm and a Probabilistic Hillclimber. This fact strongly suggests that local search algorithms are much more effective for these types of problem and that therefore they are not good candidates for comparative studies of the effectiveness of global search algorithms. The sometimes large disparity between the computational effort for *z = 0.99* and *z = 1.0* suggests that *z=0.99* figures are more reliable. The *z =1.0* figures can be skewed by a run which requires much longer than usual to obtain the target function.

## 5 CONCLUSIONS

In this paper a new method of Genetic Programming called Cartesian Genetic Programming has been presented. The chromosomes are linear strings of integers, which represent the indexed primitive functions, or the addresses in data arrays. The representation is quite generic as for a different problem one would just change the data type, leaving the genotype unchanged. The method quite naturally allows re-use of sub-functions without any explicit encoding of this. The genome has a fixed length but the coding part is completely variable up to this length, due to the presence of redundancy. Crossover can be defined as in Genetic Algorithms without any complications of ensuring a language based syntactical correctness. In this paper the method was applied to boolean concept learning, namely, even parity and 2-bit multiplier problems. It was found that these problems were best solved with an algorithm which employed a large amount of local searching, and it was shown that these methods (particularly a Probabilistic Hillclimber) were much more effective than either GP or EP. In GP crossover is thought to be very important, while in EP complicated forms of mutation are used. It appears that neither of these are required in the problem of boolean concept learning.

### References

Banzhaf W., Nordin P., Keller R. E., Francone F. D. (1998) *Genetic Programming: An Introduction*, Morgan Kaufmann.

Chellapilla K. (1998) “Evolving Modular programs without Crossover”, in *Genetic Programmimg 1998: Proceedings of the Third Annual Conference on Genetic Programming*, J. R. Koza et al (eds), Morgan Kaufmann, pp. 23-31

Chen X., and Hurst S. L. (1982) “A Comparison of Universal-Logic-Module Realizations and Their Application in the Synthesis of Combinatorial and Sequential Networks”, *IEEE. Trans. on Computers*, Vol. C-31, pp. 140- 147.

Fogel L. J., Owens A. J., Walsh M. J. (1966) *Artificial Intelligence through Simulated Evolution*, Wiley.

Fogel D. B., (1995), *Evolutionary Computation: Towards a New Philosophy of Machine Intelligence*, IEEE Press.

Iba H., Iwata M., and Higuchi T. (1996) “Machine Learning Approach to Gate-Level Evolvable Hardware”, in T. Higuchi et al (eds), *Evolvable Systems: From Biology to Hardware*, *LNCS*, Vol. 1259, Springer, pp. 327 – 343

Koza J. R. (1992) *Genetic Programming: On the programming of computers by means of natural selection*. MIT Press.

Koza J. R. (1994) *Genetic Programming II: Automatic Discovery of Reusable Subprograms*. MIT Press.

Miller J. F., Thomson P. (1998b) “Aspects of Digital Evolution: Evolvability and Architecture”, in A. E. Eiben et al (eds), *Parallel Problem Solving from Nature V*, *LNCS*, Vol. 1498, Springer, pp. 927-936.

Miller J. F., Thomson P. (1998a) “Aspects of Digital Evolution: Geometry and Learning”, in M. Sipper et al (eds), *Evolvable Systems:From Biology to Hardware, LNCS*, Vol. 1478, Springer, pp. 25-35.

Miller J. F., Thomson P., and Fogarty T. C. (1997) “Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study”, in *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*: D. Quagliarella et al. (eds), Wiley.

Poli R., (1997) “Evolution of graph-like programs with parallel distributed genetic programming”, in T. Bäck (ed), Genetic Algorithms: Proceedings of the Seventh International Conference, Morgan Kaufmann, pp. 346-353.

Sipper M., Sanchez E., Mange D., Tomassini M., Perez-Uribe A., and Stauffer A. (1997) “A Phylogenetic, Ontogenetic, and Epigenetic View of Bio-Inspired Hardware Systems”, *IEEE Trans. on Evol. Comp.*, Vol. 1, No. 1, pp. 83-97.

Teller A., Veloso M. (1995) “PADO: Learning tree structured algorithms for orchestration into an object recognition system”, Technical Report CMU-CS-95-101, Dept. of Computer Science, Carnegie Mellon University, Pittsburg, PA.