Genetic Programming for Pedestrians

Wolfgang Banzhaf

TR93-03   December 1993

Abstract

We propose an extension to the Genetic Programming paradigm which allows users of traditional Genetic Algorithms to evolve computer programs. To this end, we have to introduce mechanisms like transscription, editing and repairing into Genetic Programming. We demonstrate the feasibility of the approach by using it to develop programs for the prediction of sequences of integer numbers.

ICGA

Publication History:–

1. First printing, TR93-03, February 1993

1 Introduction

Recently, there is a surge in interest for the evolution of computer programs. The area develops into two different directions: One is the study of artificial ecologies, where computer programs compete for access to resources inside the computer, like CPU-time or memory space. This area has also been dubbed "Artificial Life" [1, 2] and leads to rich emergent phenomena, like parasitism, symbiosis, arms races between different software species. The other direction is the study of systems which evolve according to user defined behavior. This area is generally known as genetic or evolutionary programming [3, 4].

In the latter approach, one applies a sort of symbolic GA with the symbols being chosen from a set of functions and terminals. From the set of accessible functions and terminals computer programs can be composed by forming S-expressions. These symbolic programs are subsequently compiled and evaluated using given input/output pairs for the anticipated behavior of the targeted computer program or algorithm. Selection then singles out fitter programs which are reproduced and mutated for a next generation of programs.

Here we consider a system similar in function but different in the underlying mechanism. We start out with a collection of binary strings (the population) which are subsequently interpreted as computer programs. The interpretation is achieved by using a coding or transscription table specifying which binary code of a given length corresponds to which element from the set of functions and terminals available. We then proceed by evaluating and selecting programs according to their respective performance. Variation of selected binary strings results in better and better programs.

In adopting this method 1. we are able to draw from the rich experience which has accumulated over the years in GAs in general, and 2. we encounter the necessity to introduce various mechanisms also found in nature [5] for guaranteeing that the system works. Examples of those mechanisms are editing, repairing and compiling of sequences in different stages of the process, besides the transscription already mentioned.

2 The Algorithm

2.1 Basics

One can look at the basic algorithm as a kind of software production line. Programs are continuously generated by varying the available stock of binary sequences. New sequences resulting from mutation and recombination events or from outside origin translate into different programs with different behavior. Finiteness of the population causes competition between programs. A

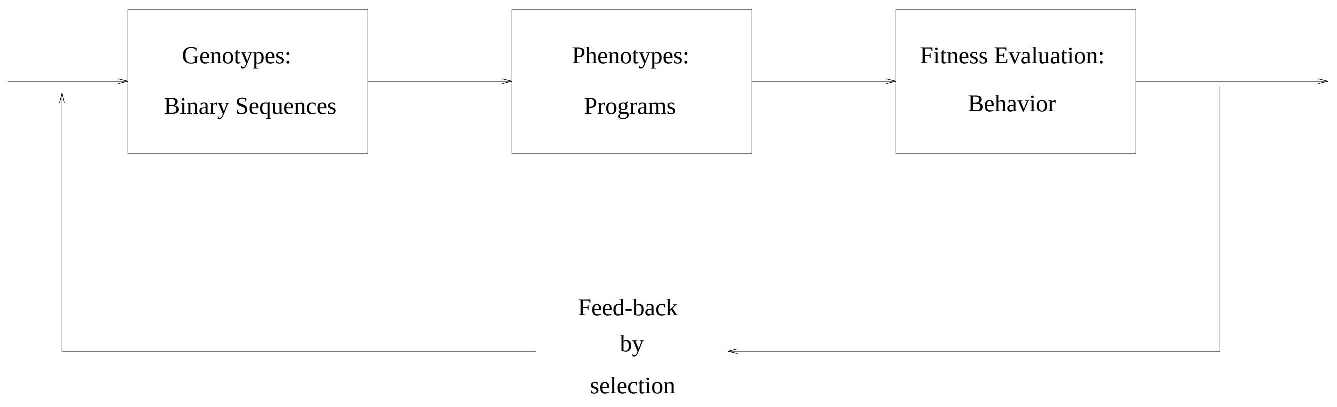

Figure 1: The Generation of programs. Random binary sequences are the input that gets iteratively selected via a feed-back mechanism. Selection is controlled by the behavioral performance of programs corresponding to bit sequences.

selection process determines which binary sequences are to be retained and reproduced for a new generation and which sequences are to be discarded (see Fig. 1).

We make use of well known genetic mechanisms like point mutation or crossover on binary strings [6, 7]. Operations actually used in our simulations will be listed below. The most important part of the algorithm, however, is the mapping from genoype (binary string) to phenotype (working program) which requires a string to produce a grammatically correct program.

Koza [3] achieves correct programs by staying in the symbolic realm of s-expression which are manipulated to yield new s-expressions. At the outset he generates valid s-expressions. In our case, a different procedure has to be applied, since initial random binary strings usually can not be considered to be working programs. Even if they were, subsequently applied operations would quickly destroy that feature of strings.

Figure 2 shows the procedure used here. Binary sequences are first transscribed into symbolic form utilizing a transscription table that states what meaning has to be assigned to a sequence of predefined length. We have used 5-bit codings of a selected number of symbolic items constituting either terminals or functions. As in Koza [3], the problem domain dictates the choice of functions and terminals.

2.2 The Problem Domain

As an exemplification of the algorithm we have chosen the problem of predicting sequences of integer numbers. Based on the information derived from few given sequence instances, the system should be able to produce the correct algorithm that enables it to predict the sequence to arbritrary length (Koza discussed in Ref. [8] a similar application).

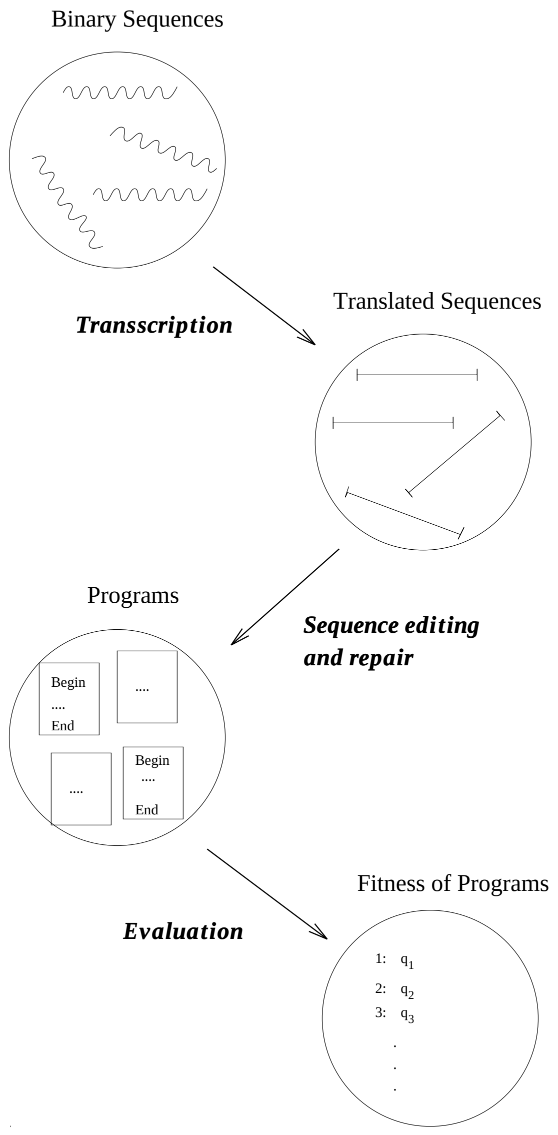

Figure 2: Mapping from genotype to phenotype. The population of binary sequences is transscribed into a high level language using a transscription table. Resulting code segments are checked on correct grammar and repaired. Sequences are edited to ensure that coded segments work properly if handed over to a compiler. Using a measure for the desirable behavior, programs are evaluated and ranked according to fitness.

Thus, if we would give the following instances to the system:

$$1, 2, 3, 4, 5 \tag{1}$$

we would expect it to learn counting and to continue with

$$6, 7, 8, 9, 10. \tag{2}$$

For the sake of generality we allow arbitrary integer sequences

$$I^{(n+1)} = f(I^{(n)}, n) \tag{3}$$

where $I^{(n+1)}$ is the follower to $I^{(n)}$, n is a counter and f is an arbitrary function.

2.3 Implementation

Programs are going to receive two integer numbers $I_0^{(\alpha)}, I_1^{(\alpha)}$ as input in each input/output example α, with $I_1^{(\alpha)}$ being the actual number in the sequence and $I_0^{(\alpha)}$ being merely a counter. As output we expect a program to produce an integer $O^{(\alpha)}$ for each instance of $\{I_0^{(\alpha)}, I_1^{(\alpha)}\}$

$$O^{(\alpha)} = g(I_0^{(\alpha)}, I_1^{(\alpha)}) \tag{4}$$

with g being a more or less complicated function encode by the corresponding binary string.

The evaluation function that shall control the selection process can now be stated as the sum square error in producing the given sequence:

$$E = \sum_{\alpha} (O^{(\alpha)} - O_0^{(\alpha)})^2 \tag{5}$$

where the target number $O_0^{(\alpha)} = I^{(n+1)}$ is identified with the follower to $I^{(n)} = I_1^{(\alpha)}$. By definition, $E \geq 0$, and we are looking for correct solutions with $E = 0$.

The terminal set T contains

$$T = \{I_0, I_1, 0, 1, 2\} \tag{6}$$

where we have included the smallest natural numbers for internal computations. The function set consists of basic operations possible with natural numbers, like addition, subtraction, multiplication, division, exponentiation, as well as the modulo and absolute value function and a comparison function returning an integer value.

$$F = \{PLUS, MINUS, TIMES, DIV, POW, ABSV, MODU, IFEQ\} \tag{7}$$

All of these functions require two arguments, with the exception of the absolute value. We have chosen to implement the default 2 argument function and to leave the second argument of the

String	*Code*				*Transscription*	*String*	*Code*				*Transscription*
0	0	0	0	0	PLUS	1	0	0	0	0	I0
0	0	0	0	1	MINUS	1	0	0	0	1	I1
0	0	0	1	0	TIMES	1	0	0	1	0	1
0	0	0	1	1	DIV	1	0	0	1	1	I0
0	0	1	0	0	POW	1	0	1	0	0	I1
0	0	1	0	1	ABSV	1	0	1	0	1	1
0	0	1	1	0	IFEQ	1	0	1	1	0	0
0	0	1	1	1	MODU	1	0	1	1	1	2
0	1	0	0	0	MODU	1	1	0	0	0	I0
0	1	0	0	1	ABSV	1	1	0	0	1	I1
0	1	0	1	0	TIMES	1	1	0	1	0	1
0	1	0	1	1	POW	1	1	0	1	1	I0
0	1	1	0	0	DIV	1	1	1	0	0	I1
0	1	1	0	1	MINUS	1	1	1	0	1	1
0	1	1	1	0	PLUS	1	1	1	1	0	0
0	1	1	1	1	IFEQ	1	1	1	1	1	2

Table 1: Transscription table of binary strings into functions and terminals. 5-bit coding shown. First bit (category bit) indicates whether a function or a terminal is coded.

absolute value function in place as a dummy variable.

Table 1 shows the 5-bit transscription of binary numbers. Notice the redundancy in coding certain functions or terminals. By randomly placing transscriptions we have used the full 5-bit set of possible codings. A technical detail is the first bit in our code which specifies whether a terminal ("1") or a function ("0") is coded.

Table 2 explains the corresponding actions taken by a program. Some of the operations needed to be modified in order to have valid numerical behavior for all possible arguments.

Function name	*Action*	*Description*
PLUS(a,b)	a + b	Integer Addition
MINUS(a,b)	a - b	Integer Subtraction
TIMES(a,b)	a * b	Integer Multiplication
DIV(a,b)	a / b	Integer Division[a]
POW(a,b)	a^b	Integer Exponentiation[b]
ABSV(a,b)	$\mid a \mid$	Absolute Value[c]
MODU(a,b)	a modulo b	Modulo Operation [d]
IFEQ(a,b)	1 if a = b , 0 otherwise	Comparison Operation

[a]Round to next integer, division through 0 results in maxvalue=1000
[b]$0^0 = 1$
[c]b is dummy
[d]b=0 returns a

Table 2: Description of functions used. Provisions have been taken for special cases.

Bitstrings are of length $L = 225$ transscribing into $N = 45$ symbols from the sets F and T. Usually, the raw transscription of a string does not fit grammar. Two requirements have to be fulfilled:

(i) The number of terminals N_T must be larger than the number of functions N_F

$$N_T > N_F \tag{8}$$

and

(ii) each sequence should start with a function call.

In order to assure these requirements we scan each string by a repair operator that corrects corresponding bits in the sequence. Resulting strings are subsequently parsed and the proper number of parentheses is inserted. Finally, the standard function structure is added leaving code that can be handed over to a compiler (see Figure 3). Note that, depending on the number of arguments supplied by the string code, a shallowly nested or a deeply nested function results.

3 Simulation Results

As mentioned before, bit strings of length 225 are used in simulations with the 5-bit code. The wildcard symbol was not allowed in our simulations.

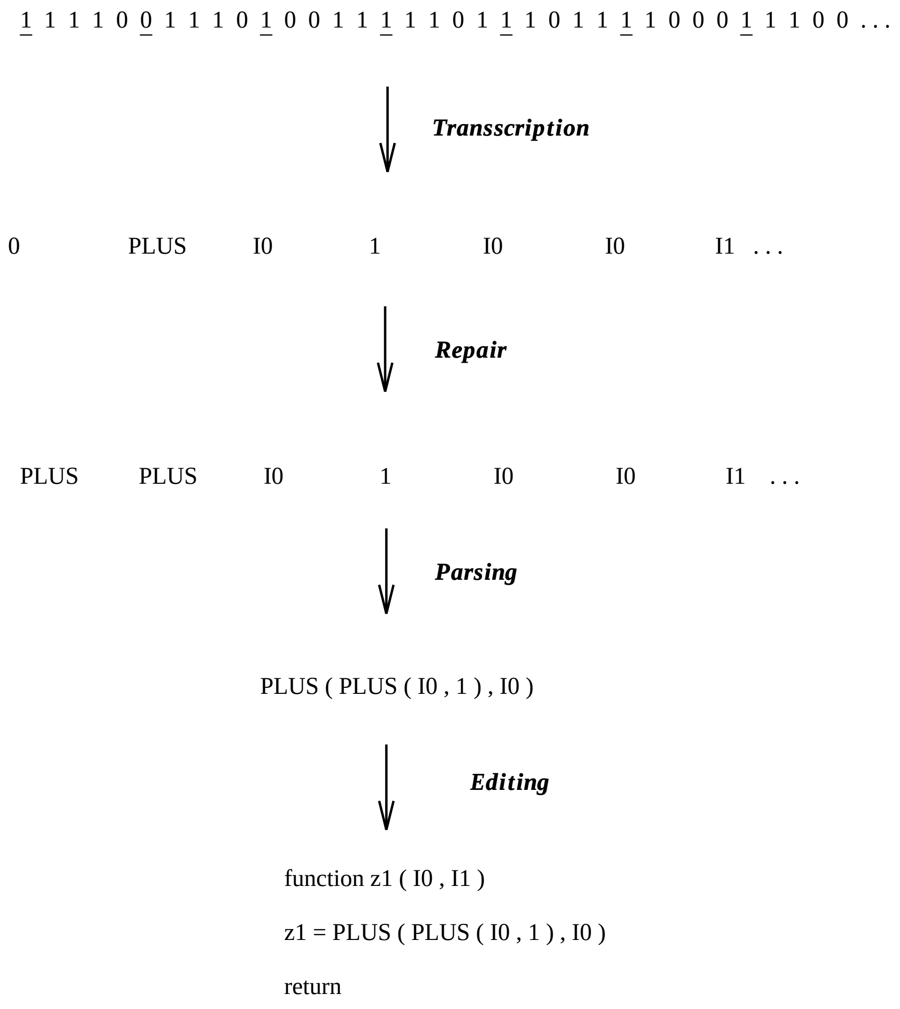

Figure 3: Typical example of the mapping from genotype to phenotype. Underlined bits are category bits determining the character of a 5-bit sequence. In the repair step, the very first bit is changed to yield a function call.

3.1 GA operators

The operators employed to constantly vary the population were taken from the following set (probabilities in parentheses):

(1) 1-bit mutation (p_1)

(2) n-bit mutation (p_2)

(3) 1-bit shift right, only category bits (p_3)

(4) 1-bit shift right, all bits (p_4)

(5) 1-bit shift left, only category bits (p_5)

(6) 1-bit shift left, all bits (p_6)

(7) 1-point crossover, only at category bits (p_7)

Three operators have been depicted schematically in Figure 4.

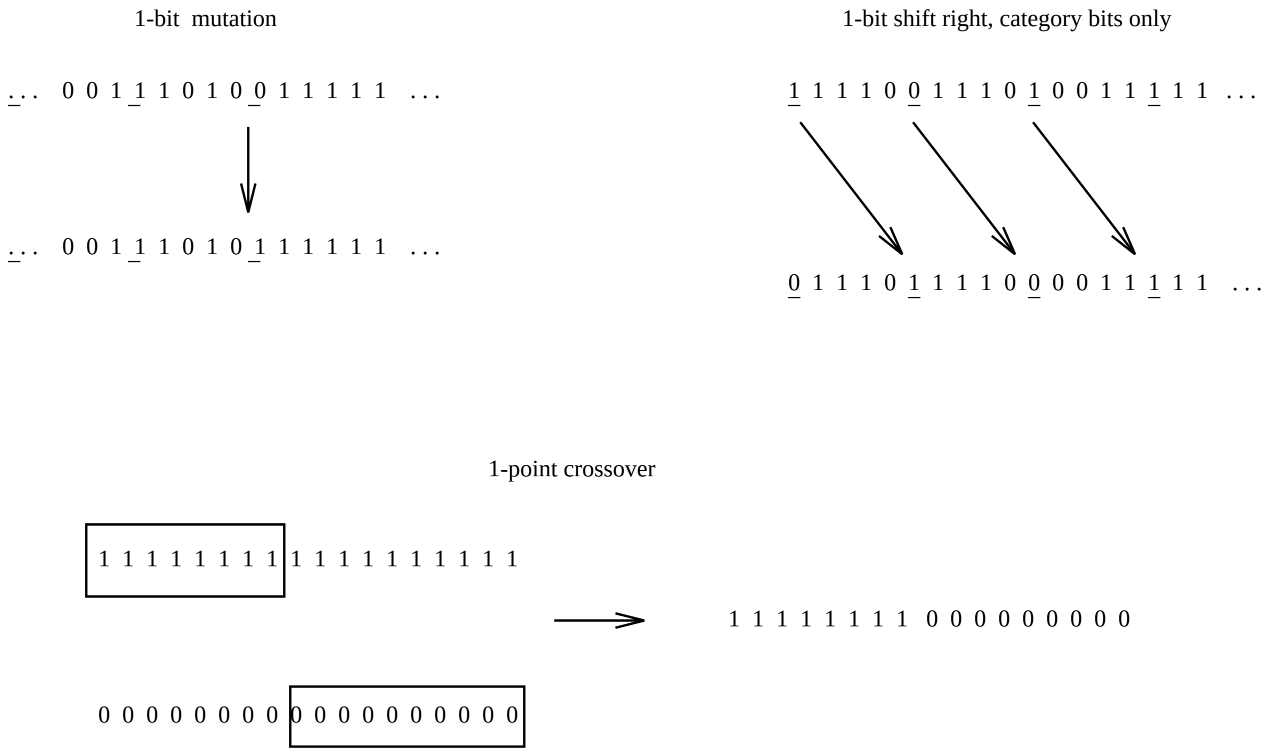

Figure 4: Three genetic operators. In 1-point crossover, the string enclosed in boxes results from the combination of 2 strings.

We performed generational selection in a population of $M = 100$ binary strings. Both, the parent generation and 100 offsprings were ranked and the 100 best were taken as a new generation of parents. The selection was constrained by the requirement that at least 50 had to consist of offspring.

3.2 Problems and general performance

Table 4 lists the 9 test problem sequences we have used for an evalution of the algorithm.

Problem	*Description*	*Number Sequence*	*Performance*
1	Count	1,2,3,4,5,6,7,8,9,10,11	very good
2	Count and jump	1,3,5,7,9,11,13,15,17,19,21	very good
3	Oscillate	2,-2,2,-2,2,-2,2,-2,2,-2,2	very good
4	Square	1,4,9,16,25,36,49,64,81,100,121	good
5	Combine (I)	3,7,13,21,31,43,57,73,91,111,133	good
6	Combine (II)	1,5,16,46,113,241,460,806,1321,2053,3056	weak
7	Dilute	1,2,5,6,9,10,13,14,17,18,21	fair
8	Faculty	1,2,6,24,120,720,5040,40320	good
9	Prime	2,3,5,7,11,13,17,19,23,29,31	weak

Table 4: General overview of problems. Performance measures are given in terms of how many different sets of operators / initial conditions led to correct solutions. Very good: almost all; good: most; fair: some; weak: none

Table 5 gives a more detailed account of the performances of different runs. The weak performance for problem 9 is reasonable as no algorithm can predict prime numbers correctly. This is not the case for problem 6, where one solution is clearly possible. In some runs, however, convergence was premature due to our use of a particular generational GAs.

O-set	*Problem*																	
	1		2		3		4		5		6		7		8		9	
A	5	0	4	7.3	4	8.8	4	23.5	3	12.7	0	-	0	-	2	16	0	-
B	5	0	5	2.8	5	1.6	4	17.8	3	30	0	-	2	33	5	9.4	0	-
C	5	1	4	5.0	5	2.2	3	26.3	3	26	0	-	1	42	5	10.6	0	-

Table 5: Performance of three different collections of operators A, B, C working on 5 different initial conditions for each problem. Each entry contains two numbers, how many cases (out of 5) resulted in correct solutions within 50 generations, and in which generation on average, the first correct solution was found. Probabilities for operators were chosen as: $A = \{p_1 = 0.9, p_7 = 0.1\}$; $B = \{p_2 = 0.7, p_3 = 0.1, p_5 = 0.1, p_7 = 0.1\}$; $C = \{p_1 = 0.7, p_4 = 0.1 p_6 = 0.1, p_7 = 0.1\}$.

Problem	*Solution*	*E*
1	PLUS(POW(0,I1),I1)	0
2	PLUS(I1,POW(2,1))	0
3	MINUS(0,TIMES(I1,1))	0
4	PLUS(PLUS(DIV(PLUS(ABSV(1,I1),I0),ABSV(1,POW(1, ABSV(I1,I0)))),I1),I0)	0
5	PLUS(ABSV(PLUS(MODU(2,I1),TIMES(I0,2)),2),I1)	0
6	ABSV(PLUS(ABSV(PLUS(I1,2),ABSV(I1,I0)),PLUS(MODU(MINUS (1,MINUS(DIV(POW(I0,2),1),2)),PLUS(2,I1)),ABSV(I1,I0))),2)	1236598
7	PLUS(TIMES(PLUS(2,TIMES(I0,IFEQ(2,MODU(0,I1)))),I0), MODU(1,TIMES(POW(I1,0),TIMES(MODU(I1,2),PLUS(0,1)))))	0
8	ABSV(PLUS(I1,TIMES(I0,I1)),DIV(ABSV(IFEQ(I1,MODU(MODU (MODU(0,TIMES(MINUS(POW(1,1),I0),PLUS(I1,PLUS(PLUS(I0, ABSV(2,I0)),DIV(I1,1))))),TIMES(I0,I1)),0)),POW(1,IFEQ (MODU(0,1),1))),I0))	0
9	PLUS(PLUS(1,2),POW(I1,MODU(1,I0)))	18

Table 6: Sample of the best solutions found for each problem. Shown are the functions that are later edited to yield programs. E is the solution quality definded in eq. 5

3.3 Discussion

Our final Table 6 lists one solution found (either correct or best) for each problem. The algorithm was quite effective in finding a path toward one of the correct solutions. If we concentrate on the failures we should look at problems 6 and 9 only. In almost all runs with problem 6, the algorithm converged prematurely leaving no chance to improve a still bad solution. We have evidence that only special sets of operators could work (run not included here).

Problem 9, on the other hand, does not have an algorithm as solution. The program we could come up with is, interestingly enough, a heuristic which adds 3 to each precedessor. Indeed, the average distance of prime numbers between 2 and 30 is 2.9. We conclude, that the program has optimized for the average distance between prime numbers which is clearly range specific.

4 Conclusion

We have shown in a specific problem instance, how the aims of genetic programming can be achieved with a usual GA. Building on the idea of genetic programming, our main extension was the mapping algorithm from binary strings representing genotypes of the population to programs representing the phenotypes. With a rather simple set of processing stages we have shown to arrive at workable programs which subsequently can undergo the competitive processes that characterize evolving populations in GAs. Given the rather graceful behavior of the algorithm, the next question to be addressed is, how to find a working transscription code for problems of various sorts.

References

[1] Langton, C.G., (ed.), *Artificial Life*, Santa Fe Institute Studies on the Sciences of Complexity, Proc. Vol. VI, Addison-Wesley, Reading, MA, 1989

[2] Ray, T.S. *Is it alive or is it GA?* Proc. 3rd Int. Conf. on Genetic Algorithms, San Diego, 1991, Belew, R.K. and Booker, L.B. (Eds.), Morgan Kauffman, San Mateo, 1991, p. 527

[3] Koza, J.R., *Genetic Programming*, MIT Press, Cambridge, MA, 1992

[4] Fogel, D.B, Atmar, W. (Eds.), Proc. 1st annual Conference on Evolutionary Programming, San Diego, 1992

[5] Lewin, B., *Genes*, Wiley, New York, 1987

[6] Holland, J.H., *Adaptation in natural and artificial systems*, University of Michigan Press, Ann Arbor, 1975

[7] Goldberg, D.E., *Genetic Algorithms in Search, Optimization & machine Learning*, Adison-Wesley, New York, 1989

[8] Koza, J.R., *Hierarchical Genetic Algorithms operating on populations of computer programs*, Proc. 11th IJCAI,1989, Morgan Kauffman, San Mateo, CA, 1989