Cartesian Genetic Programming

Julian F. Miller[1], Peter Thomson[2]

[1] School of Computer Science, University of Birmingham, Birmingham, England, B15 2TT
`j.miller@cs.bham.ac.uk`

[2] School of Computing, Napier University, 219 Colinton Road, Edinburgh, Scotland, EH14 1DJ
`p.thomson@dcs.napier.ac.uk`

Abstract. This paper presents a new form of Genetic Programming called Cartesian Genetic Programming in which a program is represented as an indexed graph. The graph is encoded in the form of a linear string of integers. The inputs or terminal set and node outputs are numbered sequentially. The node functions are also separately numbered. The genotype is just a list of node connections and functions. The genotype is then mapped to an indexed graph that can be executed as a program. Evolutionary algorithms are used to evolve the genotype in a symbolic regression problem (sixth order polynomial) and the Santa Fe Ant Trail. The *computational effort* is calculated for both cases. It is suggested that *hit effort* is a more reliable measure of computational efficiency. A *neutral search* strategy that allows the fittest genotype to be replaced by another equally fit genotype (a neutral genotype) is examined and compared with *non-neutral* search for the Santa Fe ant problem. The neutral search proves to be much more effective.

1 Introduction

In its original form Genetic Programming (GP) [10][11] has evolved programs in the form of LISP parse trees. Usually large populations are used and crossover is used as the primary method of developing new candidate solutions from older programs. In contrast to this in Evolutionary Programming [4] [5] has tended to emphasize the importance of mutation operators.

Although GP and EP place different emphasis on the evolutionary operators they both tend to share the representation of programs as parse trees in which there is no distinction between genotype and phenotype. Trees are a special form of graphs in which two nodes must have at most one path between them (a *path* is a sequence of connected nodes). One of the original motivations for a tree-based approach was to allow a solution to the problem of applying crossover to variable length genotypes. In this work a new approach is proposed called *Cartesian Genetic Programming (CGP)* where the genotype is represented as a list of integers that are mapped to directed graphs rather than trees. One motivation for this is that it uses graphs which are more general than trees. However the original motivation came from the effectiveness of the approach in learning Boolean functions where it proved to be considerably more efficient than standard GP methods [14]. In CGP the genotypes are of fixed length but the phenotypes are of variable length according to the number of unexpressed genes. The importance and potential advantages of defining a genotype-phenotype mapping were discussed by Banzhaf [2] in Binary Genetic Programming (BGP) [1]. In BGP binary strings are translated and repaired to form valid programs. In CGP no repair is necessary (see section 2). Another potential advantage of employing a genotype-phenotype mapping is that it allows for the possibility of many genotypes mapping to the same phenotype and so explicitly allows *neutrality* to be present. Neutrality refers to the presence of genotypes with the same fitness. The importance of neutrality is widely recognized in modern theories of natural molecular evolution. Indeed, Kimura [8][9] and Ohta [15][16] maintains that evolution at the molecular level is mainly due to mutations that are nearly neutral with respect to natural selection. From the point of view of fitness landscapes in artificial evolution it has been suggested that neutrality may provide a route whereby adaptive evolution may cross regions with poor fitness [7]. This view is supported in a study of a genotype-phenotype mapping that was applied to a problem of constrained optimisation using genetic programming [2]. Harvey and Thompson, in an experiment that used artificial evolution to configure a programmable silicon chip, also found that the presence of neutrality in a genetically converged population could lead to further fitness improvement [6]. It is important to note that neutrality arising from genotype redundancy can only be useful when the neutral changes are likely to alter the potential effects of future genotypic change, merely adding unexpressed genes will not facilitate useful neutral evolution.

In CGP there are very large number of genotypes that map to identical genotypes due to the presence of a large amount of redundancy. Firstly there is *node redundancy* that is caused by genes associated with nodes that are not part of the connected graph representing the program. This redundancy is very large at the beginning of the evolutionary run as many nodes are not connected in the early populations. The node redundancy gradually reduces during the run to a level that is determined by average number of nodes required to implement a satisfactory program and the maximum allowed number of nodes. Another form of redundancy in CGP, also present in all other forms of GP is, *functional redundancy*. In this case, a number of nodes implement a sub-function that actually may be implemented with fewer nodes. This growth in the number of redundant nodes constitutes *bloat*. The third form of redundancy, called *input redundancy,* occurs when some node functions are not connected to some of the input nodes. For example, in section 3 it is seen that all the nodes used to evolve programs to solve the Santa Fe Ant Trail have three inputs and one output, despite the fact that only one function uses the three inputs. Node and input redundancy both have the potential of adding useful neutrality. An unconnected node may undergo neutral change and later become connected – this might be necessary to achieve a higher fitness. A node with a redundant input might, after a mutation that altered the arity of the node function, suddenly become useful. Functional redundancy probably positively contributes to the evolvability of the target function or program as it increases the number of ways that it might be built. The possibility of disconnecting nodes that CGP allows might have a useful role in keeping the attendant bloat in check.

The principal motivations behind the work reported in this paper were twofold, firstly, to introduce CGP as an alternative methodology to standard GP, and secondly, to extend CGP to non-Boolean problems and show that it is a useful method for evolving programs with other data types. It is important that researchers in Genetic Programming explores the advantages and disadvantages of many representations rather than confining themselves to one dominant form. Diversity is important in research just as it is in evolving populations.

The organization of the paper is as follows. In section 2 the method of CGP is explained. Section 3 describes the problems that CGP is tested on and the measure used to calculate computational effort. The test problems are: symbolic regression of a sixth order polynomial from input-output data, and the harder problem of evolving a program to control an artificial ant traversing the Santa Fe Ant Trail. In section 4 the results are presented and discussed and another useful measure of computational efficiency called the *hit effort* is defined. Finally in section 5 some conclusions are given.

2 Cartesian Genetic Programming

CGP is Cartesian in the sense that the method considers a grid of nodes that are addressed in a Cartesian coordinate system. CGP has a some of points of similarity with Parallel Distributed Genetic Programming (PDGP) [17][18] and the graph-based GP system PADO (Parallel Algorithm Discovery and Orchestration) [19]. In the former graphs were evolved *without* the use of a genotype-phenotype mapping and various sophisticated crossover operators were defined. In the latter, each program is represented as an arbitrary directed graph of *N* nodes, where each node may have up to N outputs. Moreover, in PADO each node possesses its own private stack based memory, and also access to a globally defined indexed memory.

A Cartesian program (CP) denoted *P* is defined as a set $\{G, n_i, n_o, n_n, F, n_f, n_r, n_c, l\}$ where *G* represents the genotype and is itself a set of integers representing the indexed n_i program inputs, the n_n node input connections and functions, and the n_o program output connections. The set *F* represents the n_f functions of the nodes. The number of nodes in a row and column are given by n_r, n_c respectively. Finally the program interconnectivity is defined by the levels back parameter *l*, which determines how many previous columns of cells may have their outputs connected to a node in the current column (the primary inputs are treated in the same way as node outputs). In this paper only feed-forward connectivity is considered. However a Cartesian program can be readily extended to include sequential processes (e.g. loops), by allowing the inputs to nodes to be connected to the program outputs through a clock. Note that the program inputs are allowed to connect to any node input. An example of the genotype and genotype-phenotype mapping is depicted in Fig. 1 for a program with six inputs and three outputs. The example shows a feed-forward program of three rows by four columns, with l =2 and n_n = 3.

0 1 3 *0*   1 0 4 *1*   3 5 2 *2*   8 5 0 *1*   6 8 3 *2*   4 2 6 *0*
6 10 7 *2*   9 11 10 *1*   8 8 6 *2*   9 10 13 *15*   13 14 11 *16*   10 10 14 *2*   13 16 17

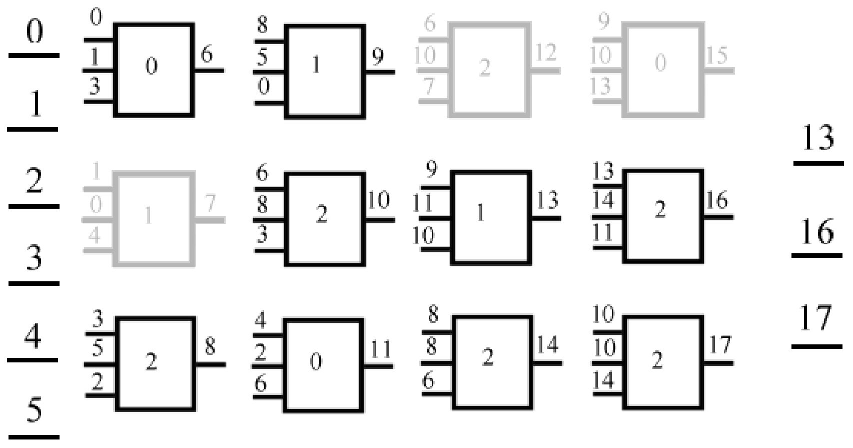

Fig. 1. Genotype-phenotype mapping a) Phenotype. b) Genotype. For a program with six inputs and 3 outputs, and three functions (0, 1, 2 inside square nodes, in italics in genotype). The grey squares indicate unconnected nodes.

Nodes in the same column are not allowed to be connected to each other, and any node may be either connected or disconnected. The CP with the most freely connected network is obtained when the n_r*=1 and l=* n_c , i.e. any node can be connected to any node on the right. The length of the genotype G that maps to a particular CP is fixed and is equal to $n_r n_c(n_n +1)+ n_o$ however this just means that the *maximum* size of the associated Cartesian program is fixed, the actual size may be anything from zero up to the maximum (in Fig. 1 only 9 nodes are connected).

When a population of genotypes is created or mutation is applied the genes must obey certain constraints in order for the genotype to represent a valid program. These are defined as follows:
Let c^n_{kj} represent the gene associated with the *k*-th input of a node in column *j* (the leftmost column is represented by 0) then it obeys the inequalities

$$\begin{gathered} c_{kj} < e_{max}\ , j < l \\ e_{min} \le c_{kj} < e_{max}\ , j \ge l \\ e_{min} = n_i + (j-l)\ n_r \\ e_{max} = n_i + j\ n_r\ . \end{gathered} \tag{1}$$

Let c^o_k represent the gene associated with the *k*-th program output then it obeys the inequalities

$$\begin{gathered} h_{min} \le c^o_k < h_{max} \\ h_{min} = n_i + (\ n_c - l)\ n_r \\ h_{max} = n_i + (\ n_c - 1)\ n_r . \end{gathered} \tag{2}$$

Let c^f_k represent the gene associated with the function of *k*-th node then it obeys the inequality

$$0 \le c^f_k < n_f\ . \tag{3}$$

Provided the genotypes obey these constraints, crossover can be freely applied to produce a valid solution.

Modern forms of GP allow the use of Automatically Defined Functions (ADFs). In the form of CGP discussed here there are no ADFs, however as node outputs may be widely re-used one can consider it as employing Automatic Re-used Outputs (AROs). One of the attractive features of CGP is that no *explicit* encoding is required to facilitate this. A potential disadvantage is that AROs are not as general as ADFs as they can only re-use an output *with the same inputs*. One can control the amount of AROs by adjusting the levels-back parameter and the number of rows and columns. AROs are most strongly favoured in a configuration of nodes with one row, and in which $l = n_c$. At the other extreme AROs are forbidden when $n_c = 1$.

In this paper two evolutionary algorithms were applied. For the symbolic regression problem a generational Genetic Algorithm (GA) was used with uniform crossover, where each gene in the offspring is randomly picked up from the parents. Size two probabilistic tournament selection was used with the winner of the tournament being accepted with a probability of 0.7 (otherwise the loser was accepted). In the Santa Fe Ant Trail problem a simple form of (1 + λ) Evolutionary Strategy (with λ ≈ 4) was used. No crossover was applied. The algorithm was as follows:

Algorithm

1. Generate initial population at random (subject to constraints)
2. Evaluate fitness of genotypes in population
3. Promote fittest genotype to new population
4. Fill remaining places in the population with mutated versions of the fittest
5. Return to step 2 until stopping criterion reached

Note that at step 3. If no new genotype has greater fitness but other genotypes have the same fitness as the best then one of these would be randomly chosen and promoted to the new population. In this paper this is referred to as *neutral* search. If only better genotypes are chosen then the search is referred to as *non-neutral*. For the Santa Fe trail both methods were examined.

3 Test problems

Two problems were studied in this paper: Symbolic regression of a sixth order polynomial [11, pages 110-122]

$$x^6 - 2x^4 + x^2 \tag{4}$$

The objective was to evolve a program that produces the given value of the sixth order polynomial above as its output, when the value of the one real-valued independent variable x is given as input. The program inputs were 1.0 and X. This departs from the definition given by Koza [11], who used *ephemeral constants* randomly defined over [-1.0, 1.0] with a given precision. Ephemeral constants are generated when the first program input is connected to a node in the initial population. From that point on in the evolutionary algorithm these constants are fixed and cannot be mutated. The function set used in this paper was *{+, - *, div}*, where *div* returns the numerator if the divisor is zero, otherwise it returns the normal result of division. The fitness cases were 50 random values of X from the interval [-1.0, +1.0]. These were fixed throughout the evolutionary run. The fitness of a program was defined as the sum over the 50 fitness cases of the sum of the absolute value of the error between the value returned by the program and that corresponding to the sixth order polynomial. The population size chosen was 10. The number of generations was equal to 8000. The crossover rate was 100% (entire population replaced by children). The mutation rate was 2%, i.e. on average 2% of the all the genes in the population were mutated. Other parameters were as follows: n_r =1, n_c =10, l =10. An evolved genotype with a high fitness is given below (with n_r =4, n_c =4, l =4):

1 0 *3*   1 1 *3*   1 1 *2*   0 0 *2*
4 4 *2*   2 1 *3*   3 4 *1*   1 4 *0*
6 3 *2*   1 8 *2*   0 8 *2*   8 6 *2*
6 11 *2*   11 11 *2*   5 3 *3*   13 13 *3*   15

Fig. 2. The genotype for a program evolved to match a sixth order polynomial. The two inputs 0 and 1 refer to 1.0 and X respectively. Functions + , - , * , div are represented by integers 0, 1, 2, 3 respectively. The program output is taken from the node with output label 15 (underlined).

The other problem studied was the Santa Fe Ant Trail [10, pages 147-155]. An imaginary ant starts at position (0,0) (top-left corner) of a 32 x 32 toroidal grid of squares. The ant initially faces east. There is a trail of food covering 144 squares with may gaps and twists and turns containing 89 pieces of food. The ant can move one square in the direction it is facing or turn left or right on the square that is situated on. Each of these actions require one time step. The ant has a sensor that enables it to detect whether there is food on the square one position ahead in the direction it is facing. The ant eats any food on squares that it is occupying. The objective is to evolve a program that can successfully navigate the ant so that it consumes all 89 pieces of food (success predicate). Only 600 time steps are allocated to execute the ant control program[1]. If a program (as is very likely) finishes before 600 time steps have elapsed then it is repeated, starting at the current state of the ant and map, this process is repeated until the 600 time steps have elapsed. This is a much more testing problem for Genetic Programming than the sixth order polynomial because of the time dependent behaviour of the ant. Thus one is evolving a sequential program with states that depend on past history. The function set was as used by Koza { *if-food-ahead, prog2, prog3*} coded as functions 0, 1, 2 respectively. The program inputs were coded as 0, 1, 2 and represented *move, left, right* respectively.

The algorithm used to evolve the Cartesian programs was described in section 2. The specific parameters used were as follows: $\lambda = 4$. A range of mutation rates per individual genotype were investigated from 4%-40%. Population size was 10, 100 runs of 12,000 generations were carried out. Other parameters were as follows: $n_r = 1$, $n_c = 20$, $l = 20$. Thus with 81 genes and a mutation rate of 10%, 8 genes would be mutated in each genotype in the population.

[1] Although it appears that Koza [10] used 400 time steps. A personal communication from William B. Langdon assured me that the correct figure was 600 and that this figure is used by other investigators.

4 Results

One method proposed for assessing the effectiveness of an algorithm, or a set of parameters was as follows [10, page 194]. 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 (meeting success predicate), 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:

$$P(M,i) = \frac{N_s(i)}{N_{total}},\ R(z) = ceil\left\{\frac{\log(1-z)}{\log(1-P(M,i))}\right\},\ I(M, i, z) = \mathrm{M}\, R(z)(i+1) \tag{5}$$

Note that when $z = 1.0$ the formulae are invalid (all runs successful). In the expression for *I(M, i, z)*, $i+1$ is used to taken in account the initial population. In this paper *z* was chosen as 0.99 so that the computational effort represented the number of evaluations required to give a probability of success of 0.99.

4.1 Sixth order polynomial

One hundred runs were carried out with 61 runs successful. The minimum computational effort (see eqn. 5) was 90,060 that corresponded to 6 runs of 1,500 generations. It is not possible to compare this with the effort computed in [11] as the experimental conditions were not the same and employing the fixed constant 1.0 conferred a great advantage over randomly chosen ephemeral constants.

4. 2 Santa Fe Ant Trail

In Tables 1 and 2 are listed the results obtained after 100 runs for various mutation rates when the neutral and non-neutral strategies were employed. The number of solutions found with the maximum fitness is listed as #hits. The minimum computational effort *I* defined in equation 5 is listed in the third column (divided by 1000). The generation at which the minimum effort was observed is shown in the fourth column. Finally the number of hits that were observed at that generation is listed in the fifth column. The first thing to notice, is that some of the figures for the computational effort are calculated for a ridiculously small number of hits (less than 10, see column 5), and thus are instantly under suspicion, since in different batches of 100 runs these figures are likely to vary enormously (Actually most of figures in Table 2 are of this type). The computational effort is plotted against mutation rate in Fig. 3. It would be very easy to pick out a very good figure for *I* and compare it favourably with results found in the literature (see Table 3). It is quite difficult to define under what conditions the calculation of computational effort is reliable without undertaking many batches of 100 runs.

Table 1. Effort to solve the Santa Fe Ant Trail (neutral)

Mutation rate (%)	#hits	Min I /1000	Generation	#hits at generation
4	34	476	420	4
6	48	280	60	1
8	68	335	2580	30
10	83	252	2100	32
12	89	235	5880	70
14	93	173	1440	32
16	86	270	9000	79
18	85	193	2760	49
20	91	212	10,620	90
25	93	209	10,440	90
30	78	331	8280	69
40	63	511	1380	12

Table 2. Effort to solve the Santa Fe Ant Trail (non-neutral)

Mutation rate (%)	#hits	Min I /1000	Generation	#hits at generation
4	17	686	300	2
6	14	831	180	1
8	20	823	360	2
10	28	139	60	2
12	30	888	5220	24
14	39	458	300	3
16	42	276	120	2
18	50	204	180	4
20	48	441	900	9
25	52	324	1620	21
30	52	413	180	2
40	40	757	840	5

Table 3. Previously published effort to solve the Santa Fe Ant Trail [12]

Method	*I/1000*
Koza GP [10, page 202]	450
Size-limited EP [3]	136
Strict Hill Climbing [12]	186
PDGP [12]	336

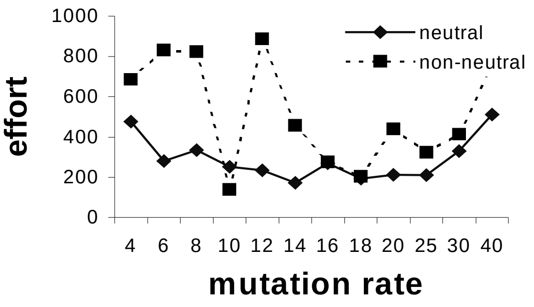

Fig. 3. The computational effort for neutral and non-neutral strategies for various mutation rates.

To assess the quality of an algorithm one could just look at the total number of hits (second column from left in Tables 1 and 2) rather than the computational effort. This is plotted against mutation rate in Fig. 4. Unfortunately this would give no information about the time taken by the algorithms to give these results.

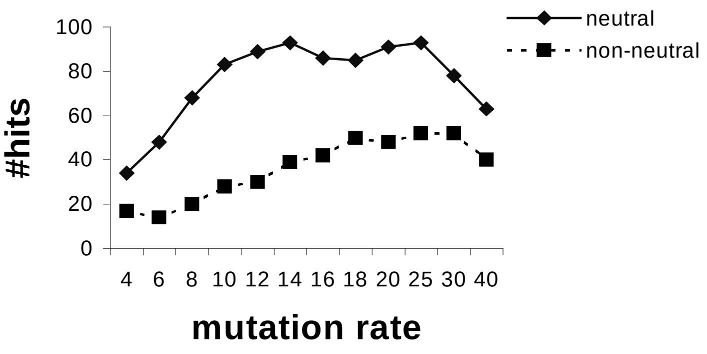

Fig. 4. The number of hits for neutral and non-neutral strategies for various mutation rates.

It is suggested that a more reliable measure of computational efficiency might be the *hit effort* which is defined here to be the total number of evaluations over the 100 runs divided by the number of hits. The is effectively the average number of evaluations required per hit. The hit effort is most reliable when a scenario is chosen that gives many successful runs as it is fairly sensitive to the number of hits. The advantage of using hit effort is that it is a figure that is measured over the *full set of runs* and is not an inference based on assumptions about likely outcome of another experiment. Qualitatively is satisfies the requirements of a good measure of computational efficiency and quality. In the experiments performed here it presented a much more regular behaviour as can be seen in Fig. 5 which shows the variation in hit effort with mutation rate.

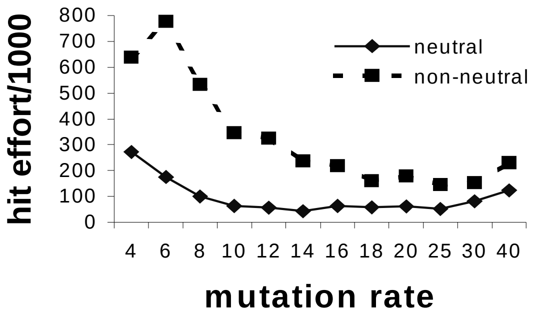

Fig. 4. The hit effort for neutral and non-neutral strategies for various mutation rates.

Langdon and Poli [12] noted that genetic programming and other search techniques do not do much better than random search and that size of the allowed programs has a marked effect on performance, with larger programs fairing more badly (above some threshold). It can be seen from the results presented here that high mutation rates of about 14% are most effective with neutral search, and that a higher rate of 25% are better for non-neutral search. These results support the findings that the fitness space associated with the Santa Fe trail has a great deal of randomness associated with it [12]. with the search space. It should also be noted that the maximum length of CG programs used in this paper was 20, so the results may be slightly better because of this fact (see [12]).

5 Conclusions

This paper has presented a new form of genetic programming called Cartesian Genetic Programming. It has already been shown to be very effective for Boolean function learning [14] and here it is extended to problems with real-valued data and time dependent behaviour. It is certainly no less effective a method than other forms of GP when tested on the Santa Fe Ant problem.

The representation of genotypes in CGP has very large redundancy and it was shown that the neutrality in fitness that this allows can be used to improve the search. An important question that needs to be answered relates to which type of neutrality is most important. Other preliminary experiments for the problem of evolving binary arithmetic functions show that the search is impaired when either the node or functional redundancy is restricted when compared to a situation in which both are allowed. Other results indicate that provided a small but sufficient amount of node redundancy is present the search is most effective. Also experiments are underway that attempt to discover the relative importance of two types of neutrality- the first, genotypes with equal fitness that map to the same phenotype and second, genotypes with equal fitness that map to different phenotypes.

It was argued that the widely used measure of computational effort [10] is unreliable, and that a new measure called hit effort should be used instead. This gave much more stable results for experiments performed on the Santa Fe Ant Trail. Although the hit effort is still sensitive to the number of successful runs, at least it is a figure that is actually calculated for the entire set of runs, rather than being an inference.

There is much more work to be done and many questions remain. Crossover is easily accomplished with the Cartesian genotypes. But does it really confer a considerable advantage for other classes of problems? The mutation mechanism employed in this paper is extremely simple. In Evolutionary Programming this is generally more sophisticated. Could more involved methods of mutation improve the performance of algorithms based on Cartesian genotypes? Why is CGP effective is it due the presence and exploitation of neutrality? Would CGP be much more effective by allowing the possibility of true Automatically Defined Functions? How could this be accomplished?

References

1. Banzhaf, W.: Genetic Programming for Pedestrains, Technical Report No. 93-03, Mitsubishi Electric Research Labs, Cambridge, MA (1993)
2. Banzhaf, W.: Genotype-Phenotype-Mapping and Neutral Variation: A Case study in Genetic Programming. In: Davidor, Y., Schwefel, H.-P., R. Männer, R. (eds.): Proceedings of the conference on Parallel Problem Solving from Nature III. Springer-Verlag, Berlin Heidelberg New York(1994) 322-332.
3. Chellapilla K.: Evolutionary programming with tree mutations: Evolving computer programs without crossover. In: Koza, J. R., et al. (eds.): Genetic Programmimg 1997: Proceedings of the Third Annual Conference on Genetic Programming. Morgan Kaufmann, San Francisco, CA. (1998) 432-438.
4. Fogel, L. J., Owens, A. J., Walsh, M. J.: Artificial Intelligence through Simulated Evolution, Wiley (1996)
5. Fogel, D. B.: Evolutionary Computation: Towards a New Philosophy of Machine Intelligence, IEEE Press (1995)
6. Harvey, I., Thompson, A.: Through the labyrinth evolution finds a way: A silicon ridge. In: Higuchi, T., Iwata, M., Liu, W. (eds.): Proceedings of First International Conference on

Evolvable Systems: From Biology to Hardware, Lecture Notes in Computer Science, Vol. 1259. Springer-Verlag, Berlin Heidelberg New York (1996) 406-422

7. Huynen, M. A., Stadler, P. F., Fontana, W.: Smoothness within ruggedness: The role of neutrality in adaptation. Proceedings of the National Academy of Science U.S.A. Vol. 93 (1996) 397-401
8. Kimura, M.: Nature, Vol. 217 (1968) 624-626.
9. Kimura, M.: The Neutral Theory of Molecular Evolution, Cambridge, Cambridge University Press (1983)
10. Koza, J. R.: Genetic Programming: On the programming of computers by means of natural selection. MIT Press (1992)
11. Koza, J. R.: Genetic Programming II: Automatic Discovery of Reusable Subprograms. MIT Press (1994)
12. Langdon, W. B., Poli, R.: Why Ants are Hard. In: Koza, J. R. et al. (eds.) Proceedings of the Third Annual Conference on Genetic Programming. Morgan Kaufmann, San Francisco, CA (1998) 193-201
13. Miller J. F., Thomson P.: Aspects of Digital Evolution: Geometry and Learning. In: Sipper, M. et al (eds.): Proceedings of the Second International Conference on Evolvable Systems: From Biology to Hardware, Lecture Notes in Computer Science, Vol. 1478. Springer-Verlag, Berlin Heidelberg New York (1998) 25-35
14. Miller, J. F.: An empirical study of the efficiency of learning Boolean functions using a Cartesian Genetic Programming Approach. In: Banzhaf, W. et al.: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99), Morgan Kaufmann, San Francisco, CA (1999) 1135-1142
15. Ohta, T.: Slightly deleterious mutant substitutions in evolution. Nature Vol. 246 (1973) 96-97
16. Ohta, T.: The nearly neutral theory of molecular evolution. Annual Review of Ecology and Systematics Vol. 23 (1992) 263-286
17. Poli, T.: Parallel distributed genetic programming. Technical report CSRP-96-15, School of Computer Science, The University of Birmingham, UK (1996)
18. Poli, R.: Evolution of graph-like programs with parallel distributed genetic programming. In: Bäck, T. (ed): Genetic Algorithms: Proceedings of the Seventh International Conference, Morgan Kaufmann, San Francisco, CA (1997) 346-353
19. Teller, A., Veloso, M.: 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 (1995)