Category: Genetic Programming
Evolution of Genetic Code on a Hard Problem

Robert E. Keller   Wolfgang Banzhaf
Systems Analysis
Computer Science Department
University of Dortmund
D-44221 Dortmund, Germany
Robert.E.Keller@WEB.DE

Abstract

In most Genetic Programming (GP) approaches, the space of genotypes, that is the search space, is identical to the space of phenotypes, that is the solution space. Developmental approaches, like Developmental Genetic Programming (DGP), distinguish between genotypes and phenotypes and use a genotype-phenotype mapping prior to fitness evaluation of a phenotype. To perform this mapping, DGP uses a genetic code, that is, a mapping from genotype components to phenotype components. The genotype-phenotype mapping is critical for the performance of the underlying search process which is why adapting the mapping to a given problem is of interest. Previous work shows, on an easy synthetic problem, the feasibility of code evolution to the effect of a problem-specific self-adaptation of the mapping.The present empirical work delivers a demonstration of this effect on a hard synthetic problem, showing the real-world potential of code evolution which increases the occurrence of relevant phenotypic components and reduces the occurrence of components that represent noise.

1 INTRODUCTION AND OBJECTIVE

Genetic programming (Koza 1992, Banzhaf *et al.* 1998) is an evolutionary algorithm that, for the purpose of fitness evaluation, represents an evolved individual as algorithm. Most GP approaches do not distinguish between a genotype, that is, a point in search space, and its phenotype, that is, a point in solution space. Developmental approaches, however, like (Keller and Banzhaf 1996, O'Neill and Ryan 2000, Spector and Stoffel 1996), make a distinction between the search space and the solution space. Thus, they employ a genotype-to-phenotype mapping (GPM) since the behavior of the phenotype defines its fitness which is used for selection of the corresponding genotype. This mapping is critical to the performance of the search process: the larger the fraction of the search space that a GPM maps onto good phenotypes, the better the performance. In this sense, a mapping is said to be "good" if it maps a "large" fraction of search space onto good phenotypes, which is captured in the formal measure of "code fitness" which is defined in (Keller and Banzhaf 1999). That work shows, on an easy synthetic problem, the effect of code evolution: genetic codes, i.e., information that controls the genotype-phenotype mapping and that is carried by individuals, get adapted such that problem-relevant symbols are increasingly being used for the assembly of phenotypes, while irrelevant symbols are less often used. This implies that the approach can adapt the mapping to the problem, which eliminates the necessity of having a user define a problem-specific mapping, which in itself is often impossible when facing a new problem, since the user does not yet understand the problem well enough. From an abstract point of view, code evolution adapts fitness landscapes, since a certain mapping defines that landscape. (Keller and Banzhaf 1999) also shows that, during evolution, it is mostly better individuals who carry better codes, and it is mostly better codes that are carried by better individuals. However, the computation of code fitness is only feasible for small search spaces, that is, easy problems, why it is of interest to test whether the effect of code evolution also takes place on a hard problem, which is the objective of this work.

First, developmental genetic programming (DGP) (Keller and Banzhaf 1996, Keller and Banzhaf 1999)

is introduced as far as needed in the context of this article, and the concept of a genetic code as an essential part of a mapping is defined. Second, the principle of the evolution of mappings as an extension to developmental approaches is presented in the context of DGP. Here, the genetic code is subjected to evolution which implies the evolution of the mapping. Third, the objective mentioned above is being followd by investigating the progression of phenotypic-symbol frequencies in codes during evolution. Finally, conclusions and objectives of further work are discussed.

2 DEVELOPMENTAL GENETIC PROGRAMMING

All subsequently described random selections of an object from a set of objects occur under equal probability unless mentioned otherwise.

2.1 ALGORITHM

A DGP variant uses a common generational evolutionary algorithm, extended by a genotype-phenotype mapping prior to the fitness evaluation of the individuals of a generation.

2.2 GENOTYPE, PHENOTYPE, GENETIC CODE

The output of a GP system is an algorithm in a certain representation. This representation often is a computer program, that is, a word from a formal language. The representation complies with structural constraints which, in the context of a programming language, are the syntax of that language. DGP produces output compliant with the syntax defined by an arbitrary context-free LALR(1) (look-ahead-left-recursive, look ahead one symbol) grammar. Such grammars define the syntax of real-world programming languages like ISO-C.A phenotype is represented by a syntactically legal symbol sequence with every symbol being an element of either a function set F or a terminal set T that both underlie a genetic-programming approach. Thus, the solution space is the set of all legal symbol sequences.

A codon is a contiguous bit sequence of $b > 0$ bits length which encodes a symbol. In order to provide for the encoding of all symbols, b must be chosen such that for each symbol there is at least one codon which encodes this and only this symbol. For instance, with $b = 3$, the codon 010 may encode the symbol a, and 2^3 symbols at most can be encoded. A genotype is a fixed-size codon sequence of $n > 0$ codons, like 011 010 000 111 with size $n = 4$. By definition, the leftmost codon is codon 0, followed by codon 1 up to codon $n - 1$.

A genetic code is a codon-symbol mapping, that is, it defines the encoding of a symbol by one or more codons. An example is given below with codon size 3.

000	001	010	011	100	101	110	111
a	b	c	d	+	*	-	/

The "symbol frequency" of a symbol in a code is the number n of occurrences of the symbol in the code, which means that n different codons are mapped onto this symbol.

2.3 GENOTYPE-PHENOTYPE MAPPING

In order to map a genotype onto a phenotype, the genotype gets transcribed into a *raw sequence* of symbols, using a *genetic code.* Transcription scans a genotype, starting at codon 0, ending at codon $n - 1$. The genotype 101 101 000 111, for instance, is mapped onto "$**a/$" by use of the above sample code.

For the following examples, consider the syntax of arithmetic expressions. A symbol that represents a syntax error at a given position in a given symbol sequence is called *illegal*, else *legal.* A genotype is mapped either onto a legal or, in the case of "$**a/$", illegal raw symbol sequence. An illegal raw sequence gets repaired according to the syntax, thus yielding a legal symbol sequence. To that end, several repair algorithms are conceivable. A comparatively simple mechanism is introduced here, called "deleting repair". Intron splicing (Watson *et al.* 1992), that is the removal of genetic information which is not used for the production of proteins, is the biological metaphor behind this repair mechanism. Deleting repair scans a raw sequence and deletes each illegal symbol, which is a symbol that cannot be used for the production of a phenotype, until it reaches the sequence end. If a syntactic unit is left incomplete, like "$a-$", it deletes backwards until the unit is complete. For instance, the above sample raw sequence gets repaired as follows: "$**a/ \rightarrow *a/ \rightarrow a/$", then a is scanned as a legal first symbol, followed by / which is also legal. Next, the end of the sequence is scanned, so that "$a/$" is recognized as an incomplete syntactic unit. Backward deleting sets in and deletes /, yielding the sequence a, which is legal, and the repair algorithm terminates. Note that deleting repair works for arbitrarily long and complex words from any LALR(1) language.

If the entire sequence has been deleted by the repair mechanism, like it would happen with the phenotype "$++++$", the worst possible fitness value is assigned

to the genotype. This is appropriate from both a biological and a technical point of view. In nature, a phenotype not interacting with its environment does not have reproductive success, the latter being crudely modeled by the concept of "fitness" in evolutionary algorithms. In a fixed-generation-size EA, like the DGP variant used for the empirical investigation described here, an individual with no meaning is worthless but may not be discarded due to the fixed generation size. It could be replaced, for instance, by a meaningful random phenotype. This step, however, can be saved by assigning worst possible fitness so it is likely to be replaced by another individual during subsequent selection and reproduction.

The produced legal symbol sequence represents the phenotype of the genotype which has been the input to the repair algorithm. Therefore, theoretically, the GPM ends with the termination of the repair phase. Practically, however, the legal sequence must be mapped onto a phenotype representation that can be executed on the hardware underlying a GP system in order to evaluate the fitness of the represented phenotype. This representation change is performed by the following phases.

Following repair, *editing* turns the legal symbol sequence into an *edited symbol sequence* by adding standard information, e.g., a main program frame enclosing the legal sequence. Finally, the last phase of the mapping, which can be compilation of the edited symbol sequence, transforms this sequence into a machine-language program processable by the underlying hardware. This program is executed in order to evaluate the fitness of the corresponding phenotype. Alternatively, interpretation of the edited symbol sequence can be used for fitness evaluation.

2.4 CREATION, VARIATION, REPRODUCTION, FITNESS AND SELECTION

Creation builds a fixed-size genotype as a sequence of n codons random-selected from the codon set. Variation is implemented by point genotype mutation where a randomly selected bit of a genotype is inverted. The resulting mutant is copied to the next generation. Reproduction is performed by copying a genotype to the next generation. An *execution probability* p of a reproduction or variation operator designates that the operator is randomly selected from the set of variation and reproduction operators with probability p. An execution probability is also called a rate. Fitness-based tournament selection with tournament size two is used in order to select an individual for subsequent reproduction or variation. Adjusted fitness (Koza 1992) is used as fitness measure. Thus, all possible fitness values exist in $[0, 1]$, and a perfect individual has fitness value 1.

3 CODE EVOLUTION

3.1 BIOLOGICAL MOTIVATION

The mapping employed by DGP is a crude metaphor of protein synthesis that produces proteins (phenotype) from DNA (genotype). In molecular biology, a codon is a triplet of nucleic acids which uniquely encodes one amino acid, at most. An amino acid is a part of a protein and thus corresponds to a symbol. Like natural genotypes have evolved, the genetic code has evolved, too, and it has been argued that selection pressure works on code properties necessary for the evolution of organisms (Maeshiro 1997).Since artificial evolution gleaned from nature works for genotypes, the central hypothesis investigated here is that artificial evolution works for genetic codes, too, producing such codes that support the evolution of good genotypes.

3.2 TECHNICAL MOTIVATION

In DGP, the semantics of a phenotype is defined by its genotype, the specific code, repair mechanism and semantics of the employed programming language. Especially, different codes mean different genotypic representations of a phenotype and therefore different fitness landscapes for a given problem. Finally, certain landscapes differ extremely in how far they foster an evolutionary search. Thus, it is of interest to evolve genetic codes during a run such that the individuals carrying these codes find themselves in a beneficial landscape. This situation would improve the convergence properties of the search process. A related aspect is the identification of problem-relevant symbols in the F and T sets. In order to investigate and analyze the effects of code evolution, an extension to DGP has been defined and implemented, which will be described next.

3.3 INDIVIDUAL GENETIC CODE

DGP may employ a *global code*, that is, all genotypes are mapped onto phenotypes by use of the same code. This corresponds to the current situation in organic evolution, where one code, the standard genetic code, is the basis for the protein synthesis of practically all organisms with very few exceptions like mitochondrial protein synthesis.

(Keller and Banzhaf 1999) introduces the algorithm of genetic-code evolution. If evolution is expected to oc-

cur on the code level, the necessary conditions for the evolution of any structure must be met. Thus, there must exist a structure population, reproduction and variation of the individuals, a fitness measure, and a fitness-based selection of individuals. A code population can be defined by replacing the global genetic code by an *individual code*, that is, each individual carries its own genetic code along with its genotype. During creation, each individual receives a random code. An instance random code is shown:

000	001	010	011	100	101	110	111
*	/	*	a	a	d	+	a

Note that a code, since it is defined as an arbitrary codon-symbol mapping, is allowed to be redundant with respect to certain symbols., i.e., it may map more than one codon onto the same symbol. This is not in contradiction to the role of a code, since also a redundant code can be used for the production of a phenotype. Actually, redundancy is important, as the empirical results will show.

3.4 VARIATION, REPRODUCTION, CODE FITNESS AND SELECTION

A point code mutation of a code is defined as randomly selecting a symbol of the code and replacing it by a different symbol random-selected from the symbol set. Point code mutation has a certain execution probability. Reproduction of a code happens by reproducing the individual that carries the code. The same goes for selection.

4 EMPIRICAL ANALYSIS

The announced major objective of the present work is to empirically test whether the effect of code evolution takes place on a hard problem, i.e., whether the codes are adapted in a problem-specific way that is beneficial to the search process. To this end, a run series is performed on a hard synthetic problem. Evolution means a directed change of the structures of interest, which are, in the present case, the genetic codes of the individuals. In the context of the present work, the phenomenon of interest is the change of the symbol frequencies of the target symbols. If the effect of code evolution takes place on a hard problem, this must show as a shift of symbol frequencies such that the resulting codes map codons on relevant symbols rather than on other symbols.

According with the objective of the present work, a hard problem has to be designed, and problem-relevant as well as irrelevant symbols, which represent noise, have to be contained in the symbol set. Note that the objective is not to solve the problem but to observe code evolution during the DGP runs on the problem. There are several conditions for a problem that is hard for an evolutionary algorithm, and one of the most prominent ones is that the search space is by many orders of magnitude larger than the set of individuals generated by the algorithm during its entire run time.

The problem to be considered is a symbolic function regression of an arithmetic random-generated function on a real-valued parameter space.

All function parameters come from $[0, 1]$, and the real-valued problem function is given by

$$f(A, B, a, b, .., y, z) = j+x+d+j*o+e*r-t-a+h-k*u+a-k-s*o*i-h*v-i-i-s+l-u*n+l+r-j*j*o*v-j+i+f*c+x-v+n-n*v-a-q*i*h+d-i-t+s+l*a-j*g*v-i-p*q*u-x+e+m-k*r+k-l*u*x*d*r-a+t-e*x-v-p-c-o-o*u*c*h+x+e-a*u+c*l*r-x*t-n*d+p*x*w*v-j*n-a-e*b+a.$$

Accordingly, the terminal set used by the system for all of its runs is given as $\{A, B, a, b, .., y, z\}$, and the four parameters A, B, y, z do not occur in the expression that defines the problem function, that is, they represent noise in the problem context. In order to provide for noise in the context of the function set, too, this set shall be given as $\{+, -, *, /\}$. As the division function / does not occur in the expression that defines the problem function, it represents noise. As only 5 symbols, – i.e., about 15% –, of all 32 symbols represent noise, identifying those by random is unlikely.

Due to the resulting real-valued 28-dimensional parameter space, a fitness case consists of 28 real-valued input values and one real output value. Let the training set consist of 100 random-generated fitness cases. A population size of 1,000 individuals is chosen for all runs, and 30 runs shall be performed, each lasting for exactly 200 generations. That is, there is no run termination when a perfect individual is found so that phenomena of interest can be measured further until a time-out occurs after the evolution of the 200th generation.

As there are 32 target symbols, the size of the codons must be set to five, at least, in order to have codes that can accomodate all symbols, and for the run series, the size is fixed at five. As $2^5 = 32$, the space of all possible genetic codes contains 32^{32} elements, or approximately $1.5 * 10^{48}$ codes, including 32! or about $2.6 * 10^{35}$ codes with no redundancy. Genotype size 400 is chosen, i.e., 400 codons make up an individual genotype, while the length of the problem func-

tion, measured in target symbols, is about 200. This over-sizing of the genotype size strongly enlarges the search space, making the problem at hand very hard. As the codon size equals five and the genotype size equals 400, the search space contains 2^{400*5} individuals, or 10^{602}, and as the single-bit-flip operator is the only genotypic variation operator, this corresponds to a 2000-dimensional search space. According to the experimental parameters, $6 * 10^6$ individuals are evaluated during the run series, so that the problem search space as well as the space of all codes are significantly larger than the set of search trials, that is, individuals, generated by the approach.

The execution probabilities are 0.85 for genotype reproduction, 0.12 for point genotype mutation, and 0.03 for point code mutation. Note that the point code mutation rate is only 25 percent of the genotype point mutation rate. This has been set to allow the approach to evolve the slower changing codes by use of several different individuals that carry the same code, like genotypes are evolved by use of several different, usually static, fitness cases. We hypothesize that these differing time scales are needed by the approach to distinguish between genotypes and codes.

The codes of the individuals of an initial generation are randomly created, so that each of the 32 symbol frequencies is about one in generation 0.

5 RESULTS AND DISCUSSION

Subsequently, "mean" refers to a value averaged over all runs, while "average" designates a value averaged over all individuals of a given generation.

Top down, figure 1 shows the progression of the mean best fitness and the mean average fitness.

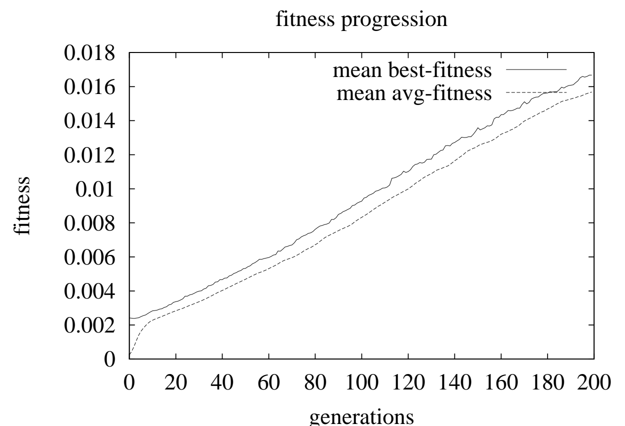

Figure 1: Top down, the curves show the progression of the mean best fitness and mean average fitness.

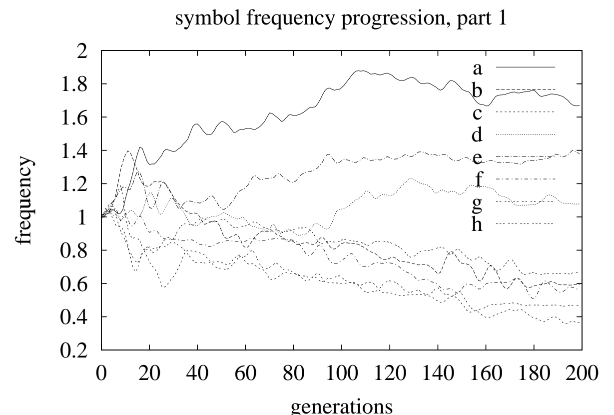

Figure 2: Progression of the mean symbol frequency in the code population.

Both curves rise, indicating convergence of the search process, which is relevant to the hypothesized principle of code evolution that is given below.

The following four figures together illustrate the progression of the mean symbol frequencies for all 32 symbols, while each figure, for reasons of legibility, displays information for eight symbols only.

As for the interpretation of figures 2 to 5, the frequency value F for a symbol S in generation G says that, over all runs, S occurs, on average, F times in a genetic code of an individual from G. As there are 32 positions in each code, F theoretically comes from $[0, .., 32]$, while practically the extreme values of the range will not be reached due to point code mutation. A value below one indicates the rareness of S in most codes of the generation, while a value above one signals redundancy of S, that is, on average, more than one codon of a genotype gets mapped onto S, or, put differently, S gets more often used for the build-up of a phenotype. Note that, due to the random creation of the codes for generation 0, all curves in all figures approximately begin in $(0, 1)$, since there are 32 codes and 32 positions in each code.

A general impression to be gained from all figures is that, after an initial phase of strong oscillation of the frequencies, the frequency distribution stabilizes. This phenomenon is typical for learning processes in the field of evolutionary algorithms, where after an initial exploratory phase a phase of exploitation sets in. It can be observed for fitness progressions, where well-performing individuals are of interest, and it can also be oberved for the presented symbol-frequency distributions, where a beneficial genotype-phenotype map-

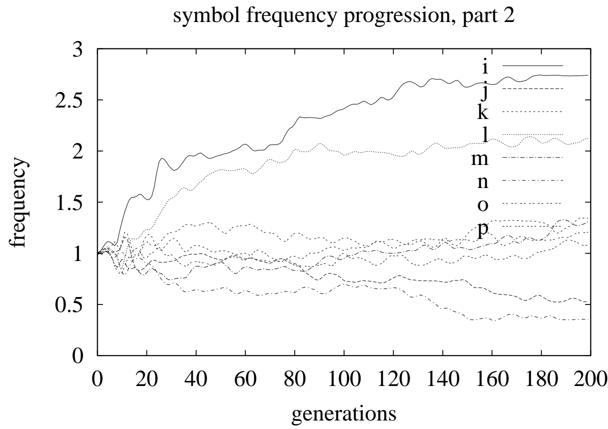

Figure 3: Progression of the mean symbol frequency in the code population.

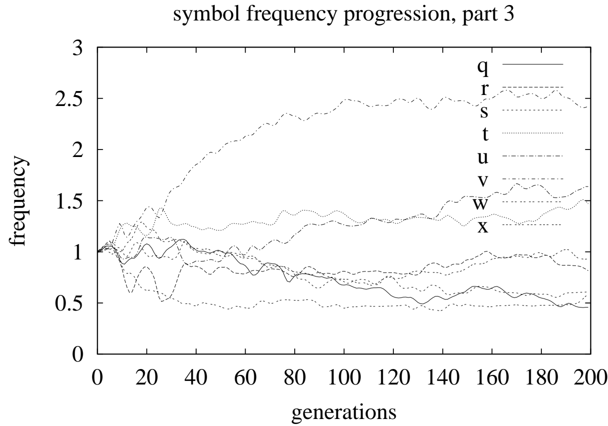

Figure 4: Progression of the mean symbol frequency in the code population.

ping is of interest.

Specifically, the figures show a classification of the symbols with respect to their relevance for the solving of the problem, as will be argued next. Due to initial oscillation, more reliable results are to be gained from late generations, which is why the frequencies of the final 200th generation shall be considered. In order to accomodate for variance of the mean average frequency values, symbols with a frequency of 0.8 or lower shall be designated as clearly under-represented in number in the genetic codes. As levels of statistical significance mostly come from $[0.9, .., 0.99]$, 0.8 represents a safe upper threshold for insignificance.

These symbols are $A, B, b, c, f, g, h, j, n, q, s, w, y, /$, which implies that four of five, that is, 80%, of the noise-representing symbols $A, B, y, z, /$ are under-

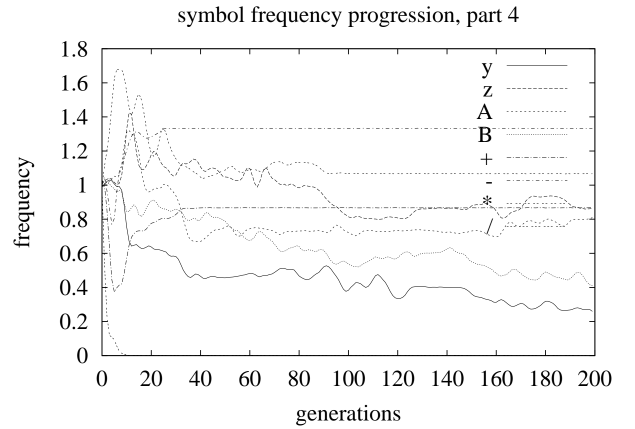

Figure 5: Progression of the mean symbol frequency in the code population. Note that the arithmetic-operator frequencies stabilize very fast and stay very stable. This is not an artefact.

represented, while 63% of the problem-relevant symbols, that is, 17 of 27 symbols, are represented with a frequency of one and higher.

The frequency of a symbol in a code heavily influences the frequency of the occurrence of the symbol in the phenotype onto which a genotype carrying the code is mapped. Thus, if non-noise symbols do and noise symbols do not become elements of the phenotype, this situation increases the likelihood that the phenotype has an above-average fitness. Therefore, the presented result represents the objective of the present work, as it verifies that the effect of code evolution also takes place on a hard problem in a way beneficial to the search process.

As for the principle of code evolution, we hypothesize that,for a certain problem, some individual code W, through a point code mutation, becomes better than another individual code L. Thus, W has a higher probability than L that its carrying individual has a genotype together with which W yields a good phenotype. Therefore, since selection on individuals is selection on codes, W has a higher probability than L of being propagated over time by reproduction and being subjected to code mutation. If such a mutation results in even higher code fitness, then the argument that worked for W works for W's mutant, and so forth.

6 CONCLUSIONS

It has been shown empirically that the effect of code evolution works on a hard problem, that is, genetic

codes carried by individuals get adapted such that, during run time, problem-relevant phenotypic symbols are increasingly being used while irrelevant symbols are less often used.

7 FUTURE RESEARCH

Several hypotheses must be investigated, among them the claim that DGP with code evolution outperforms non-developmental approaches on hard problems. We argue especially that there is a high potential in code evolution for the application to data-mining problems, since, in this domain, a "good" composition of a symbol set is typically unknown since the functional relations between the variables are unknown due to the very nature of data-mining problems. We hypothesize that code evolution, through generation of redundant codes, enhances the learning of significant functional relations by biasing for problem-specific key data and filtering out of noise. Last not least, the hypothesized principle of code evolution, that is, the co-operative co-evolution of individuals and codes, shall be investigated.

References

Banzhaf, Wolfgang, Peter Nordin, Robert E. Keller and Frank D. Francone (1998). *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, dpunkt.verlag.

Keller, Robert E. and Wolfgang Banzhaf (1996). Genetic programming using genotype-phenotype mapping from linear genomes into linear phenotypes. In: *Genetic Programming 1996: Proceedings of the First Annual Conference* (John R. Koza, David E. Goldberg, David B. Fogel and Rick L. Riolo, Eds.). MIT Press, Cambridge, MA. Stanford University, CA. pp. 116–122.

Keller, Robert E. and Wolfgang Banzhaf (1999). The evolution of genetic code in genetic programming. In: *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference, July 13-17, 1999, Orlando, Florida USA* (W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela and R.E. Smith, Eds.). Morgan Kaufmann. San Francisco, CA.

Koza, John R. (1992). *Genetic Programming: On the Programming of Computers by Natural Selection.* MIT Press, Cambridge, MA.

Maeshiro, Tetsuya (1997). Structure of Genetic Code and its Evolution. PhD thesis. School of Information Science, Japan Adv. Inst. of Science and Technology. Japan.

O'Neill, M. and C. Ryan (2000). Crossover in grammatical evolution: A smooth operator?. In: *Genetic Programming* (Riccardo Poli et al., Ed.). Number 1802 In: *LNCS.* Springer.

Spector, Lee and Kilian Stoffel (1996). Ontogenetic programming. In: *Genetic Programming 1996: Proceedings of the First Annual Conference* (John R. Koza, David E. Goldberg, David B. Fogel and Rick L. Riolo, Eds.). MIT Press, Cambridge, MA.. Stanford University, CA. pp. 394–399.

Watson, James D., Nancy H. Hopkins, Jeffrey W. Roberts, Joan A. Steitz and Alan M. Weiner (1992). *Molecular Biology of the Gene.* Benjamin Cummings. Menlo Park, CA.