Y. Davidor, H.-P. Schwefel and R. Männer (Eds.)
Springer, Berlin, 1994
pp. 322 — 332

Genotype-Phenotype-Mapping and Neutral Variation — A case study in Genetic Programming

Wolfgang Banzhaf
Department of Computer Science, Dortmund University
Baroper Str. 301, 44221 Dortmund, GERMANY
banzhaf@tarantoga.informatik.uni-dortmund.de

Abstract. We propose the application of a genotype-phenotype mapping to the solution of constrained optimization problems. The method consists of strictly separating the search space of genotypes from the solution space of phenotypes. A mapping from genotypes into phenotypes provides for the appropriate expression of information represented by the genotypes. The mapping is constructed as to guarantee feasibility of phenotypic solutions for the problem under study. This enforcing of constraints causes multiple genotypes to result in one and the same phenotype. Neutral variants are therefore frequent and play an important role in maintaining genetic diversity. As a specific example, we discuss Binary Genetic Programming (BGP), a variant of Genetic Programming that uses binary strings as genotypes and program trees as phenotypes.

1 Introduction

Historically, there is a long dispute among evolutionary biologists as to the main engine for evolutionary change. Is selection the force that overwhelmingly forms the outcome of evolution? Or is the more influential force that of variation, i.e. events of mutation or recombination of genetic material that provide for the continued progress in evolving generations? Starting with Darwin [1], who already acknowledged the existence of variations selection is not working against, the dispute raged back and forth, with Kimura's neutrality theory of evolution [2, 3] being the most prominent expression of the idea of a variation engine of evolution. Paraphrased, Kimura's theory states that evolution at the molecular level is mainly due to mutations that are nearly neutral with respect to natural selection. Mutation and a resulting random drift of genomes are thus considered main forces behind evolution. Kimura notes that this continuous variation of genetic material, with most of it being neither advantageous nor disadvantageous, is key in understanding natural genetic diversity.

It has become possible in recent years to look in detail at the molecular level of evolution, i.e. the genotypic level where the effects of this neutrality assumption are to be expected. And, indeed, a high variety of genotypes have been shown to exist [4]. In some way, it seems that the mapping from genotypes to phenotypes allows many different genotypes to result in phenotypes of comparable functionality.

The central idea of the present paper is to take this kind of genotype-phenotype-mapping (GPM) into the area of artificial evolution, as it is applied in Evolutionary Computation paradigms like genetic algorithms (GAs), evolutionary strategies or evolutionary programming. We shall be providing ample possibilities for the evolution of neutral variants, since our GPM will by construction be such that many genotypes will map into one phenotype. This is due to the fact that the optimization problem considered is of the class of constrained optimization problems [5]. What we shall not do here, however, and this must be a subject for further study later on, is to consider the important problem of transformation of function. We suspect that in order to include this aspect, an extension of the GPM has to be made that is able to model developmental processes.

The treatment of constrained optimization problems is different from that of unconstrained problems, since a candidate solution is not only judged according to its fitness or quality but also has to obey certain restrictions which exclude entire regions of solution space as unfeasible. Therefore, in constrained optimization two sometimes antagonistic criteria have to be satisfied, (i) quality and (ii) feasibility of a solution. Very often, constraining the solution space leads to local hills or valleys which are difficult to overcome with traditional methods of optimization.

There are several ways to treat constraints in GAs [6] — [8]: One is to include a penalty term in the fitness function. This is called a soft constraint and it is equivalent to allowing the solution to break a constraint by trading solution quality against penalty. In the long run, trading gets more and more difficult until, finally the optimization can only progress by implementing the constraint completely. The other method in order to find good and feasible solutions is a hard constraint. This means to start out with feasible solutions and to restrict search operations in such a way that only feasible solutions are generated during the evolutionary search. Thus, only specially adapted operators may be used that are shown to guarantee feasibility of a transformed solution.

Though in the latter case all solutions are guaranteed to be feasible, the restriction of search operations might lead to solutions that are not optimized. Vice versa, in the former case it might happen that quality improvement by *not* obeying constraints is so large that it is not possible to return to a feasible solution once the algorithms has entered the region of good (but unfeasible) solutions.

A third method based on constraint programming has been proposed to treat constrained optimization problems [9] which makes use of state space search that is enhanced by knowledge of the problem domain.

Another way to optimize problems under constraints is the GPM method proposed here. Search space and solution space are separated, and a mapping between search space (genotypes), where unrestricted search operators can be applied and solution space (phenotypes), where feasibility of solutions is guaranteed, is introduced. Whereas any genotype is allowed in search space, an appropriate GPM provides for feasible solutions in solution space only. It is immediately clear that this requires a mapping of search regions that would lead to otherwise unfeasible solutions, into feasible solutions.

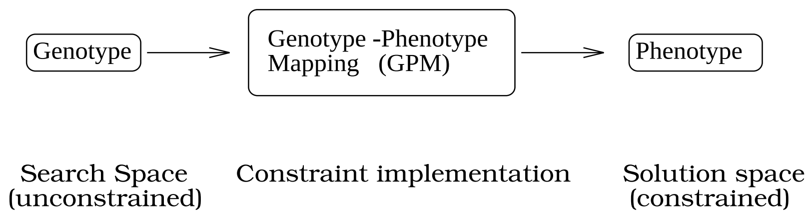

Fig. 1. Functionality of different parts of the algorithm. Unrestricted genotypes scan the search space, GPM provides for an implementation of constraints, solutions are represented by phenotypes in the restricted solution space.

The method used for enforcing feasibility is a two-step process, the first step of which is a raw mapping, with the second being a correction in case the feasibility test is negative. If we now consider a genotype that leads to a feasible solution without being corrected in the second step, we can think of variations that do not change its quality but only remove feasibility. It is those genotypes that will be corrected back by the second step into the nearest feasible phenotype. In other words, some of the information this genotype is carrying is ignored due to the GPM. Needless to say that precisely these are the set of neutral variants that do not influence in any visible way the performance of a phenotypic solution.

The point we are going to make is that the high variability of neutral variants allowed due to GPM permits the algorithms to escape local optima on saddle surfaces. In high dimensional spaces this is the way out of local trapps, and neutral variants are the only way out — apart from appropriate recombination or a lucky but improbable event of just jumping over the barrier. As Eigen [10] has emphasized, a random drift due to the generation of neutral variants broadens the population distribution sufficiently as to secure an escape route with manyfold probability (compared to the fast-decaying spherical distribution around the consensus sequence).

We are going to study one specific example of this phenomenon in Genetic

Programming [11], a variant of GAs.

2 Genetic Programming

Genetic Programming (GP) is a variation on the theme of applying artificial selection to structures that are to be optimized [11]. In the context of algorithms, programming languages are the material from which structures are build. Since any kind of language obeys certain grammatical rules, constraints have to be followed by expressions in the respective language.

The GP approach established by Koza starts out from the tree representation of a program. As is well known each expression in a context-free formal language can be stated as a hierarchical tree of nodes of different arity. Arity 0 nodes (leaves) are symbols from a predetermined terminal set. Nodes with arity ≥ 1 are symbols from another set, called functions, which carry a number of arguments. The semantics of these symbols has to be provided by an outside user of the GP system. GP manipulates these symbols by operating on trees and subtrees in order to generate variations or recombinations of already existing trees. The continuous selection of improved trees may lead to algorithms which treat given input/output pairs (fitness cases) correctly. At the outset, the system is seeded with a collection of randomly generated trees, that are guaranteed to obey certain parameter settings (e.g. depth of the tree). The trees are interpreted and evaluated according to the respective choice in functions and terminals, and generate, depending on input data, a certain behaviour in form of algorithmic output.

Because of the presence of grammatical constraints in a programming language, GP is a natural test bed for the ideas developed in Section 1. Figure 2 compares the genotype-phenotype mapping found in Nature with a generic model and an adaption to the needs of evolving algorithms which we call Binary Genetic Programming (BGP). As in Koza's approach, fitness evalution in algorithms is done by comparison of the required output with the actual output of the algorithms.

Whereas Koza evolves phenotypes (program trees) that behave as programs when interpreted with the appropriate system, we evolve genotypes (bit strings). In the spirit of the first method to treat constrained problems mentioned above, in Koza's work only those search operators are used that allow to produce valid program trees. In contrast, we can use any kind of search operator working on bit-strings, since it is the subsequent mapping into program trees, that guarantees fulfillment of the constraints. In order to arrive at correct programs we follow the GPM method of Figure 2c. The main ingredients to this mapping are, first, a coding of pieces of the bit-string into the nodes of a program tree, i.e. into members of the set of functions and terminals, and, second, a correction mechanism that is able to check statements and to transform them into the nearest correct statement if an error was detected. Thirdly, constant pieces of

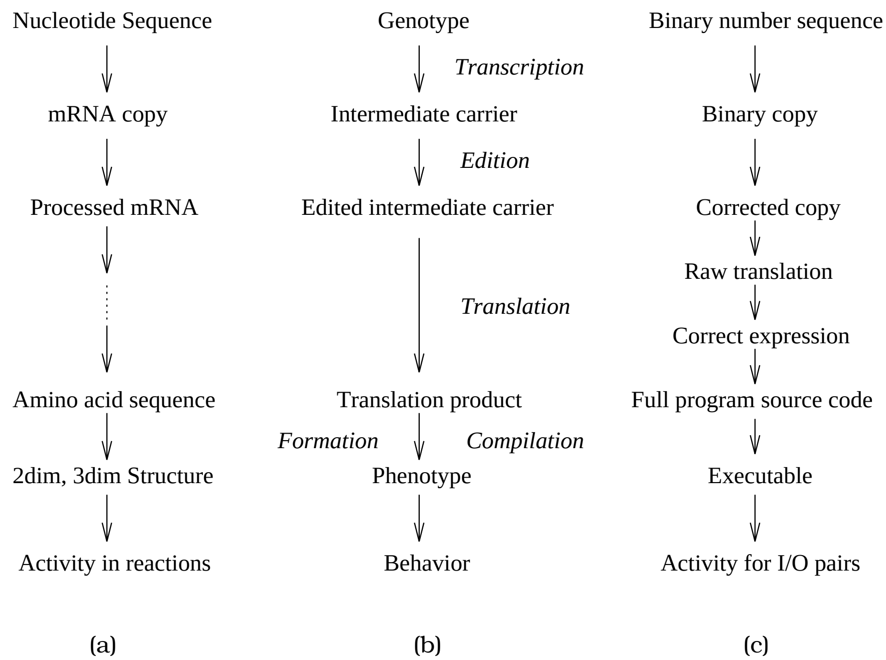

Fig. 2. Sketch of genotype – phenotype mapping; (a) in Nature, (b) generic model, (c) in Binary Genetic Programming (BGP).

code are added in order to arrive at a working program which can be compiled. The role of the intermediate carrier is to be able to cut out bits encoding parameters without being forced to permanently remove this information from the bit string. A comparable role exists for messenger RNA (mRNA) in the natural process of expressing genotypes [12], though these mechanisms ar emuch more complex in the natural system than those used here.

One additional twist that GPM gives to a Genetic Programming system is that it allows for varying random numbers. This is due to the freedom that an expression of information gives during the mapping process. We do not intend to compare this new functionality with the more inflexible approach Koza is taking but rather introduce it here as a sideline.

3 Some details of the BGP implementation

GPM expresses genetic information carried by a bit-string in two phases. Phase I takes a bit-string and generates a high-level language construct that is used in Phase II by a regular compiler to generate relocated machine instructions. Whereas Phase II is standard, some details have to be given for Phase I in order to understand the approach.

After drawing a copy from the original bit-string, this copy is processed by scanning it from left to right in 5-bit sections. Each 5-bit section is considered to be a code for a symbol from Table 1 giving either a non-terminal symbol (left side) or a terminal symbol (right side). As we can see, there is a certain amount of redundancy in the coding, with the selection of particular codes for particular nodes rather arbitrary. As in Koza's approach, a decision has to be made as to what sort of functions and terminals are to be used in the application at hand. For the regression application we discuss here, we have chosen numerical functions as the set of non-terminals. On the terminal side we use X, the input to our regression, and $R1 \ldots R4$, random numbers generated in different intervals.

Code	*Symbol*	*Code*	*Symbol*	*Code*	*Symbol*	*Code*	*Symbol*
0 0 0 0 0	PLUS	0 1 0 0 0	PLUS	1 0 0 0 0	X	1 1 0 0 0	X
0 0 0 0 1	MINUS	0 1 0 0 1	MINUS	1 0 0 0 1	R1	1 1 0 0 1	R1
0 0 0 1 0	TIMES	0 1 0 1 0	TIMES	1 0 0 1 0	X	1 1 0 1 0	X
0 0 0 1 1	THRU	0 1 0 1 1	THRU	1 0 0 1 1	R2	1 1 0 1 1	R2
0 0 1 0 0	POW	0 1 1 0 0	POW	1 0 1 0 0	X	1 1 1 0 0	X
0 0 1 0 1	SI	0 1 1 0 1	EX	1 0 1 0 1	R3	1 1 1 0 1	R3
0 0 1 1 0	CO	0 1 1 1 0	RLOG	1 0 1 1 0	X	1 1 1 1 0	X
0 0 1 1 1	EX	0 1 1 1 1	RLOG	1 0 1 1 1	R4	1 1 1 1 1	R4

Table 1. Transcription 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.

The generation of random numbers deserves more discussion: If we would generate a different random number every time, e.g. $R1$ is used in a program, no reliable function could be developed around this random number. Koza solves this problem by defining "random ephemeral constants". These are constants that are generated once and for all at the creation of a program tree. Later they are only combined into different trees, with their value kept fixed. Our approach is different, since we have decided to represent random numbers directly on the bit-string. We use two procedures, one applying intron coding of random numbers, the other applying category bit coding. The former procedure is as follows: Once the scanning process discovers one of the R string-codes, it cuts out the following 2 5-bit sections, i.e. the next 10 bits, for use as a random number. These bits are interpreted as a natural number between 0 and $2^{10} - 1$ and mapped into an interval according to Table 2. The latter method makes use of the fact, that we can look at all the category bits of a string as another (though shorter) bit-string, which might be interpreted differently. Thus, when the code calling for a random number is discovered, the next ten category bits are used to generate the natural number mentioned above. Further treatment is equal in both cases.

The advantage of an explicit representation of random numbers on strings is that their value now becomes susceptible to random mutations and other variations due to genetic operators that are impossible in Koza's scheme. In effect, they have become parameters coded on the string. This additional feature comes at a cost, though. A further processing step of interpreting parameter bits has to be introduced.

Symbol	*Interval*	*Sort*
R1	[-1,+1]	Real
R2	[0,1]	Real
R3	[-10,+10]	Integer
R4	[0,10]	Integer

Table 2. Treatment of numbers cut out from the string.

After a string has been scanned and optionally processed, a raw translation of its information is generated which is analyzed for grammatical correctness. If there are not enough terminals supplied on the string, for instance, category bits are changed from right to left until enough terminals are present. The resulting translation is parsed and supplied with parentheses and other necessary characters as well as with unchanging program headers and tails in order to arrive at valid source code of a target language like C or FORTRAN. Finally, a commercial compiler takes over and produces an executable that is run for the predetermined (preferably large number of) test cases of input/output pairs. Due to the fact that we can use compiled code, execution is by orders of magnitude faster than interpreted LISP-code.

4 Numerical simulation for selected regression problems

For this contribution we examined the following two regression problems:

$$y_1 = \frac{1}{2}x^2 \tag{1}$$

$$y_2 = e^{-3x^2} \tag{2}$$

We used 500 fitness cases in the range $x \in [0, 4]$. Fitness was defined as the degree to which the above function is approached by an individual from the population. We use the inverse measure of the quadratic deviation

$$e = \sum_{i=1,500} e_i = \sum_{i=1,500} (f(x_i) - y_{1,2})^2 \qquad (3)$$

and try to minimize it. An individual is considered successful if it is able to approach $y_{1,2}$ within a small interval for all the fitness cases:

$$e_i \leq 0.01 \;\; \forall i. \qquad (4)$$

If the first such individual appears, the population as a whole is considered successful and the simulation is stopped.

Figure 3a-b is the cumulative success probability for 50 runs each. Compared are the two different approaches to implementing random numbers as parameters on the string. From the figures we can see that the category-bit approach is generally better than the intron approach. Also, eq. (1) was not so easily solvable within 50 generations.

As an example, we give solutions for eq. (2) that were achieved by continuing a run that would have stopped for the statistical measurements:

$$f(x) = 0.05^{x^2} \qquad (5)$$

Note that $e^{-3} = 0.0498$. Another solution was

$$f(x) = e^{x^{x^{-3}}} \qquad (6)$$

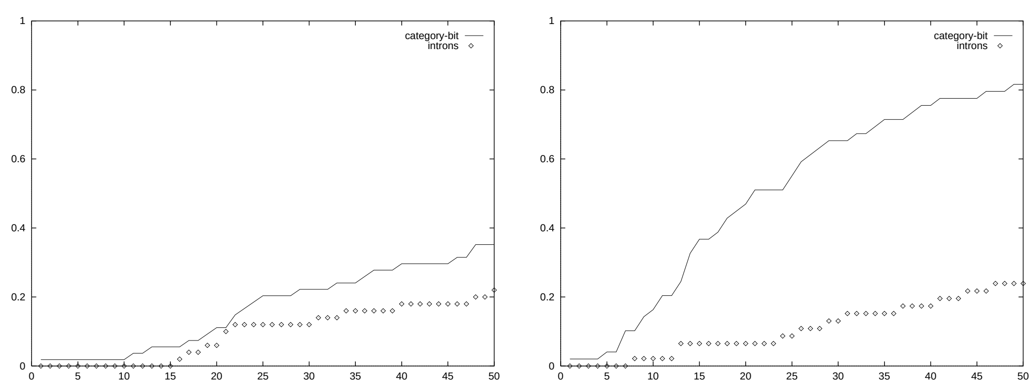

Fig. 3. Success probability for 50 runs of problem (a) eq. (1) and (b) eq. (2).

5 Analysis and conclusion

We shall now take one typical run and analyse it with respect to the question of neutral variations. As a natural distance metric in our search space we take the Hamming distance. Figure 4 - 6 show histograms of Hamming distances from all genotypes to the best genotype. In Figure 4, left, we see that distances are distributed initially around 120 bits, which reflects the fact that the zero-th generation does not yet have any preference in search space (strings have 225 bits).

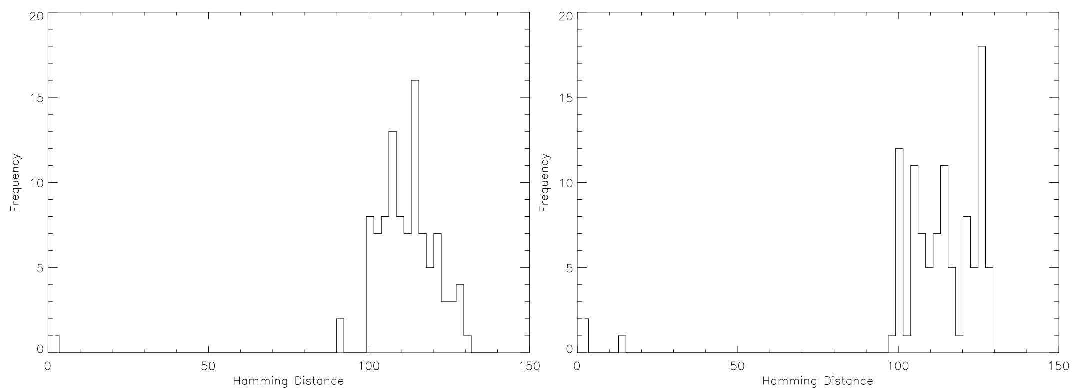

Fig. 4. Histogram of Hamming distances. Left: Generation 0; Right: Generation 7.

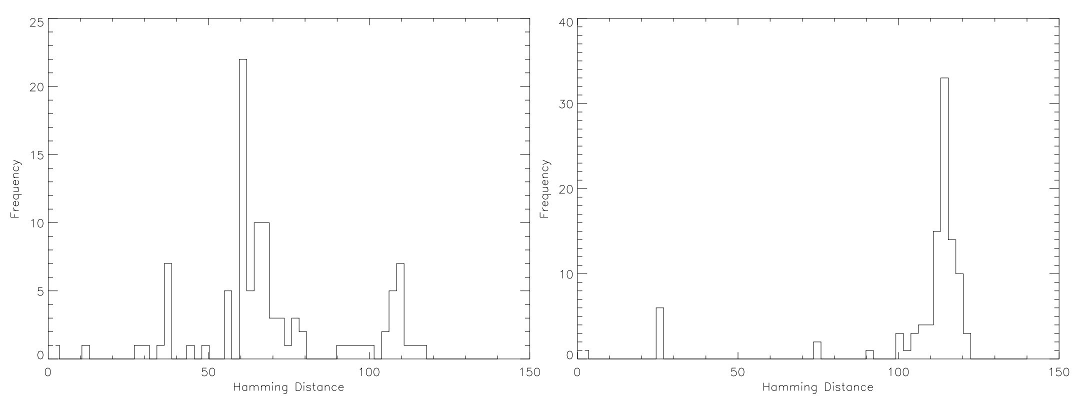

Fig. 5. Histogram of Hamming distances. Left: Generation 15; Right: Generation 19.

The average Hamming distance is smaller in generation 7, since a movement

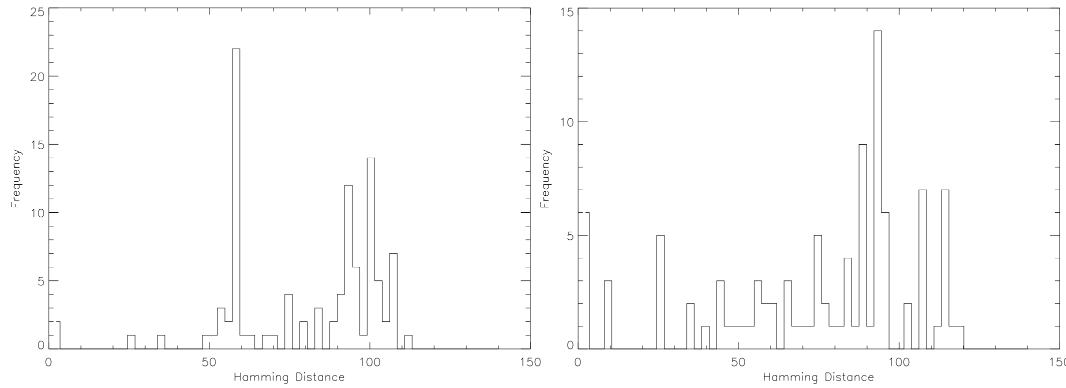

Fig. 6. Histogram of Hamming distances. Left: Generation 22; Right: Generation 25.

towards better solutions has begun. In generation 15 (Figure 5,left), shortly before a correct solution is discovered, the population has an average Hamming distance of appr. 65, concentrating genotypes in more promising regions of the search space. Then a big jump occurs for the best individual, effectively restoring the original distribution of distances.

Figure 6 showns a broadening of the distribution as more and more individuals move into the correct region. Though the deviation e of all individuals is now within 2×10^{-3} of the optimal value, $e = 0$, diversity of genotypes is still remarkable. The reason for this behavior lies in the fact, that part of the genotypic information is not expressed in the phenotype. The unexpressed part may vary arbitrarily, without any consequences for the fitness of the individual.

It is interesting to note further, that program trees that are derived from fixed length binary strings have varying length. Like other bits, the category-bit of random strings is 1 with probability $\frac{1}{2}$. For a translation into binary trees as we use them in our BGP, a function will be at place 1 with certainty. At place 2, a terminal will follow with probability $p_2 = \frac{1}{2}$. For the tree to be complete, a terminal should follow on place 3. This happens with probability $p_3 = \frac{1}{4}$. Thus, one quarter of all randomly generated trees will be very short. It follows that one eighth will contain 2 functions, and so on. A natural tendency toward shorter program trees is therefore built into the algorithm.

The GPM method introduced here is useful in other constraint optimization problems as well. It is hypothesized that the strict separation of search operations (in genotype space) and constraint implementation (in GPM) allows systems of this sort to work more flexible than systems that work with phenotypes only.

References

1. C. Darwin: On the Origin of Species by Means of Natural Selection. London: Murray 1972, 6th Edition
2. M. Kimura: Nature 217, 624 - 626 (1968)
3. M. Kimura: The Neutral Theory of Molecular Evolution. Cambridge: Cambridge University Press 1983
4. T. Mukai: Experimental Verification of the Neutral Theory. In: T. Ohta, K. Aoki (Eds.): Population Genetics and Molecular Evolution. Berlin: Springer 1985
5. G. Reklaitis, A. Ravindran, K. Ragsdell: Engineering Optimization Methods and Applications. New York: Wiley 1983
6. D. Orvosh, L. Davis: Shall we repair? Genetic Algorithms, combinatorial optimization, and feasibility constraints. In: S. Forrest (Ed.): Proc. 5th Int. Conference on Genetic Algorithms, ICGA-93. San Mateo: Morgan Kaufmann 1993
7. H. Fang, P. Ross, D. Corne: A promising GA approach to Job-Shop scheduling, rescheduling, and Open-Shop scheduling problems. In: S. Forrest (Ed.): Proc. 5th Int. Conference on Genetic Algorithms, ICGA-93. San Mateo: Morgan Kaufmann 1993
8. R. Nakano: Conventional GA for job shop scheduling. In: R. Belew, L. Booker (Eds.): Proc. 4th Int. Conference on Genetic Algorithms, ICGA-91. San Mateo: Morgan Kaufmann 1991
9. J. Paredis: Exploiting Constraints as Background Knowledge for Genetic Algorithms: a Case-study for Scheduling. In: R. Männer, B. Manderick (Eds.): Parallel Problem Solving from Nature, 2. Amsterdam: Elsevier Science Publishers 1992
10. M. Eigen: Steps toward Life: a perspective on evolution. Oxford: Oxford University Press 1992
11. J.R. Koza: Genetic Programming. Cambridge (USA): MIT Press 1992
12. M.W. Gray, P.S. Covello: RNA editing in plant mitochondria and chloroplasts. FASEB J. **7** (1993) 64