The Evolution of Genetic Code in Genetic Programming

Robert E. Keller
Systems Analysis
Computer Science Department
University of Dortmund
D-44221 Dortmund, Germany
keller@icd.de

Wolfgang Banzhaf
Systems Analysis
Computer Science Department
University of Dortmund
D-44221 Dortmund, Germany
banzhaf@icd.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 problem-specific manually designed genetic code, that is a mapping from genotype components to phenotype components. The employed genetic code is critical for the performance of the underlying search process. Here, the evolution of genetic code is introduced as a novel approach for enhancing the search process. It is hypothesized that code evolution improves the performance of developmental approaches by enabling them to beneficially adapt the fitness landscape during search. As the first step of investigation, this article empirically shows the operativeness of code evolution.

1 INTRODUCTION

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, 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 search space that a GPM maps onto good phenotypes, the better the performance. Therefore it is of interest to examine whether a developmental approach can evolve GPMs along with the ultimately interesting phenotypes, which is the objective of this article.

First, developmental genetic programming (DGP) (Keller and Banzhaf 1996) is introduced as far as needed in the context of this article. The concept of a genetic code as an essential part of the employed GPM is defined. Second, the principle of the evolution of GPMs 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 GPM. Third, the operativeness of this principle is demonstrated on an easy artificial problem so that the process of code evolution can be made more transparent. Finally, further research, especially real-world applicability of code evolution, is 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 GPM prior to the fitness evaluation of each 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	+	*	-	/

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 complex mechanism, called "replacing repair", is presented in (Keller and Banzhaf 1996), which replaces an illegal symbol by a computed legal symbol. 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.

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 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 GENETIC-CODE EVOLUTION

3.1 BIOLOGICAL MOTIVATION

GPM 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. In order to investigate and analyze the feasibility of code evolution, an extension to DGP has been defined and implemented, which will be described next.

3.3 INDIVIDUAL GENETIC CODE

So far, DGP variants used 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.

If evolution is expected to occur on the code level, the necessary conditions for the evolution of any structure must be met. 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 could receive a random code. Actually, for the empirical runs, a user-defined code is supplied during creation for experimental reasons to be explained later. 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. 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. Indeed, 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. This corresponds to a simple concept of "quality" of a code: since a code carried by an individual defines the fitness of the individual's phenotype, this fitness is a naive definition for the fitness of the individual's code. However, the same code, if carried by another individual with a different genotype, is likely to result in a different phenotypical fitness when used for the mapping of the genotype. Thus, a finer measure for code fitness is needed.

For the following empirical investigation in the context of an easy artificial problem, a code-fitness measure based on search space enumeration and the knowledge of a perfect solution is defined. The use of this measure is prohibitive in the face of a real-world problem due to the associated search space size and often unknown existence and structure of a perfect solution. Code fitness of a given code is defined as the fraction of the search space that is mapped on a perfect solution under control of the code and the repair mechanism. For instance, if the search space contains 2^{12} genotypes and a given code maps 200 genotypes on a perfect solution, the code fitness is about 0.05.

4 HYPOTHESIS

The hypothesis to be investigated in this article is that code evolution in terms of code fitness works, that is, the best and average code fitness rise over time.

We argue that, for a certain problem, some individual code W, through a point code mutation, may have gained a higher code fitness 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. As a consequence, the average code fitness should rise along with the average individual fitness.

5 EMPIRICAL ANALYSIS

A DGP run series using individual genetic codes is performed on an easy artificial problem so that code fitness can be computed in acceptable time. In order to test the hypothesis, the means of best and average code fitness and best and average individual fitness are measured. Also, the frequencies of the symbols occurring in the codes are measured, which allows an observation of code redundancy. Note that it is *not* an objective of this article to test whether a GP approach using code evolution can solve a problem or whether code evolution enhances the search performance of an approach. For the time being, the objective is to test the hypothesis given in text unit 4, that is, code evolution works, at all.

The problem is a symbolic function regression of a known function on a four-dimensional parameter space. The function is $f(a, m, v, q) = a * a$. Three further parameters m, v, q are introduced for noise generation. All parameter values shall be real-valued and come from $[0, 1]$. Due to the resulting real-valued four-dimensional parameter space, a fitness case consists of four real input values and one real output value. The training set consists of 100 random-generated fitness cases.

A population size of 50 individuals is chosen for all runs and 50 runs are performed. Each run lasts for exactly 50 generations. These relatively small values seem appropriate considering the simple regression problem. Especially, there is no run termination when a perfect individual is found so that the ongoing code evolution can be measured further until a time-out occurs after the evolution of generation 49.

$\{m, v, q, a, +, *, -, /\}$ serves as symbol set so that the shortest perfect phenotype "$a * a$" can be represented. 3-bit codons are used which implies that a code maps 2^3 codons. As there are 8 symbols in the symbol set, the code space contains 8^8 or approximately $10^{7.2}$ codes, including 8! codes with no redundancy. Genotype size 4 is chosen, so that the only perfect phenotype that can be evolved in the described setup is "$a * a$". As the codon size equals 3, the search space contains 2^{4*3} or approximately $10^{3.61}$ individuals, so that the codon space is significantly larger than the genotype space. This is beneficial with respect to the empirical focus since we concentrate on code evolution.

The execution probabilities are 0.6 for reproduction, 0.32 for point mutation, and 0.08 for point code mutation. Note that the individual mutation rate is over 50 percent of the reproduction rate and point code mutation is only 25 percent of the individual mutation rate. This has been set to allow the DGP system 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 fitness cases. We hypothesize that these differing time scales are needed by the evolutionary learning process to distinguish between genotypes and codes.

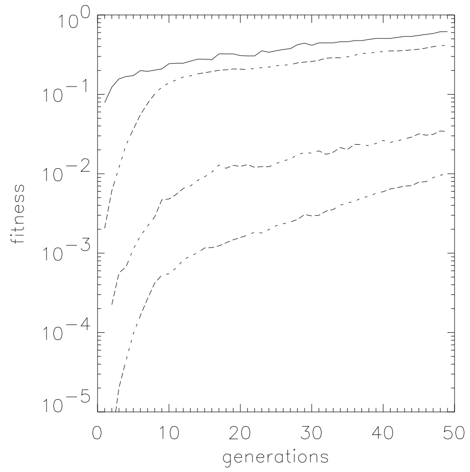

Figure 1: Top down the curves show the progression of the mean best fitness, mean average fitness, mean best code fitness, mean average code fitness on a logarithmic fitness scale.

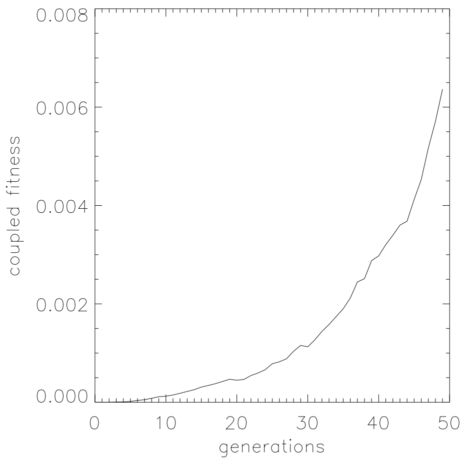

Figure 2: The mean coupled fitness is shown.

The codes of the individuals of an initial generation are *not* randomly created but set such that each codon encodes the symbol '$-$'. This way, all initial codes are identical and have code fitness zero as there is no possible genotype in the search space that gets mapped onto the perfect phenotype "$a * a$" by deleting repair. On the contrary, all initial genotypes get mapped onto the same raw sequence "$- - - -$" which results in the worst possible individual fitness, that is 0, for the associated phenotypes. That way, no initial code has a selective advantage over another code, and the same goes for all initial genotypes. Additionally, genotype evolution and the hypothesized code evolution start under worst possible conditions.

6 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, mean average fitness, mean best code fitness, and mean average code fitness on a logarithmic fitness scale. As explained, the resulting individual fitness and code fitness values are zero in generation 0. Thus, due to the logarithmic fitness scale, the two individual-fitness graphs start at generation 1, as $log(0)$ is not defined. Both curves rise, indicating convergence of the search process.

Convergence is slowed by the combination of tournament selection, which is not an elitist selection variant, and of the high individual mutation rate in relation to the reproduction rate. Due to this combination, evolved good individuals may get lost again before they reproduced sufficiently to get propagated by selection and reproduction. The loss of evolved perfect individuals is observed in the experiments.

Due to the small code mutation rate, the two code-fitness curves start later, at generation 2. Convergence can be observed, supporting the hypothesis that code evolution works in principle.

The question is raised if better individuals tend to have better codes and vice versa. To approach this topic, coupled fitness is defined as the product of the fitness and the code fitness of an individual. In the cases of bad individuals having bad code and bad individuals having good code, averaged coupled fitness is low. Only in the case of individual and code quality rising together, averaged coupled fitness rises. Figure 2 illustrates the coupled-fitness progression. The rising graph indicates that indeed better individuals tend to have better codes, backing the hypothesis that code evolution works by propagation of those codes that define better individuals which in turn propagate their codes.

Figure 3 illustrates the progression of the mean symbol frequency in the code population over time. The initially high frequency of the minus signs declines as other symbol frequencies rise during exploration of the code search space. Finally, the relatively high frequencies for the symbols a and $*$, composing the perfect phenotype, emerge. Especially, more and more codes become redundant on symbol a, which prevails in the perfect phenotype, which can be seen from the mean frequency 1.5 for a in generation 49. Put differently,

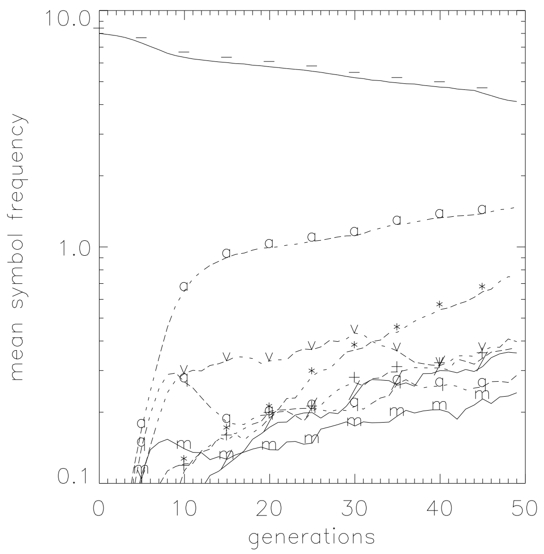

Figure 3: For each symbol and generation n, the mean number of occurrences of the symbol in all codes of all generations n over all runs is shown on a logarithmic scale.

the system learns the significance of a and $*$, while it recognizes the insignificance of the other symbols introduced as noise.

A particular run that went over 200 generations produced an a-redundancy of 2.9 and a $*$-redundancy of 1.3 in generation 199 as top redundancies over all 8 symbols. From this run, three evolved codes of good or perfect individuals and their code fitnesses follow.

aaa/a-	0.156250
*aaaaa+a	0.158203
aaa/**	0.184570

7 CONCLUSIONS

The evolution of genetic code has been introduced to genetic programming, and it has been shown empirically that code evolution works on an artificial problem, that is, the best and average code fitness rise over time.

8 FUTURE RESEARCH

The performances of non-developmental GP and DGP with and without code evolution will be compared on harder problems, in particular on real-world problems. We argue especially that there is a high potential in code evolution for the application to data-mining problems. 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. Code evolution also has a potential for solving dynamical problems, since a representation change through code evolution may help the search process to keep up with a changing distribution of local optima in the search space.

Acknowledgments

We are grateful for constructive comments of anonymous reviewers. R.K. acknowledges support from the GEPROG project, 01 IB 801.

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.

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.

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.