Explicit Control of Diversity and Effective Variation Distance in Linear Genetic Programming

Markus Brameier and Wolfgang Banzhaf

Department of Computer Science, University of Dortmund,
44221 Dortmund, Germany
brameier,banzhaf@LS11.informatik.uni-dortmund.de

Abstract. We have investigated structural distance metrics for linear genetic programs. Causal connections between changes of the genotype and changes of the phenotype form a necessary condition for analyzing structural differences between genetic programs and for the two objectives of this paper: (i) Distance information between individuals is used to control structural diversity of population individuals actively by a two-level tournament selection. (ii) Variation distance is controlled on the effective code for different genetic operators – including a mutation operator that works closely with the applied distance metric. Numerous experiments have been performed for three benchmark problems.

1 Introduction

In contrast to other evolutionary search algorithms, like evolution strategies (ES), genetic programming (GP) may fulfill the principle of *strong causality*, i.e., small variations in genotype space imply small variations in phenotype space [12], only weakly [14]. Obviously, changing just a small program component may lead to almost arbitrary changes in program behavior. However, it seems to be intuitive that the more instructions are modified, the higher is the probability of a large fitness change.

The *edit distance*, sometimes referred to as *Levenshtein distance*, [6] between varying length character strings has been proposed as a metric for representations in genetic programming [9,13]. Such a metric not only permits to analyze genotype diversity within the population but offers a possibility to investigate the effect (*step size*) of variation operators. In [7] correlation between edit distance and fitness change of tree programs has been demonstrated for different variation operators and test problems.

This work applies the edit distance metric to operate selectively on *representative substructures* of the program representation used in linear GP (LGP). Correlation between structural and semantic distance as well as distribution of distances are documented for two different types of variation. One type uses recombination while the other one is based on (macro) mutations only.

The major objective of this contribution is to control structural diversity, i.e., the average program distance, in LGP populations *explicitly*. Therefore, we introduce a two-level tournament that selects for fitness on the first level and for diversity on the second level. In the course of these experiments development of both diversity and prediction performance are analyzed. We will see that prediction improves significantly if the diversity level of a population is increased.

The simplest form of diversity control might be to seed randomly created individuals regularly into the population during runtime. In [9] a more explicit maintenance of diversity is proposed by creating and seeding individuals that fill "gaps" of under-represented areas in genotype space. However, experimental evidence is not given for this rather complicated and computationally expensive approach. Until now, explicit diversity control is a rarely investigated technique in genetic programming. Recently, de Jong *et al.* [8] could improve parsimony pressure through Pareto-selection of fitness and tree size by adding a (third) diversity objective. A more *implicit control* of genetic diversity, by comparison, offer semi-isolated sub-population, called *demes*, that are widely used in the area of evolutionary computation (see e.g. [16]).

The second objective of this paper refers to the structural distance between a parent program and its offspring, i.e., the variation distance. The change induced by a variation operator on the effective, i.e., fitness-relevant, code may differ significantly from the amount of absolute change. By monitoring the *effective* variation distance, structural step sizes may be controlled more precisely in relation to the effect on program semantics. We will see that even strong restrictions of the maximum allowed mutation distance do not necessarily restrict freedom of variation.

2 Basics on Linear GP

Programs in tree-based genetic programming (TGP) denote expressions from a functional programming language like LISP [10]. In linear genetic programming (LGP) [1], instead, the program representation consists of variable-length sequences of instructions from an imperative programming language like machine code [11] or C [3]. *Operations* manipulate variables (*registers*) and constants and assign the result to a destination register, e.g., $r_i := r_j + 1$. Single operations may be skipped by preceding *conditional branches*, e.g., $if(r_j > r_k)$.

The imperative program code is divided into *effective* and *non-effective* instructions where only the effective code may influence program behavior (see program example printed in Section 5). The non-effective instructions are referred to as *introns*. This separation of instructions results from the linear program *structure* – not from program execution – and can be computed efficiently during runtime [3].

We distinguish two different variants of linear GP in this work. While the standard approach applies recombination by crossover to vary program length the other approach works with mutations exclusively. The *linear crossover* operator exchanges two arbitrarily long sub-sequences of instructions between two individuals. If the operation cannot be executed because one offspring would exceed the maximum length crossover is performed with equally long sub-sequences. *Macro mutations* include deletions or insertions of single (full) instructions here and represent an alternative growth operator to crossover. *Micro mutations* change the smallest program components that comprise a single operator, a register or a constant.

In [4] we report on *effective mutations* which guarantee explicitly that the effective code is altered. This reduces the probability that a mutation stays neutral in term of a fitness change. If an instruction is inserted its destination register is chosen in such a way that the instruction is effective at the corresponding program position.

3 Distance Metrics for LGP Programs

The *string edit distance* [6] operates on arbitrarily sequences of characters. It measures the distance between two strings by counting the number of basic operations – including insertion and exchange of single elements – that are necessary to transform one string into another. The string edit distance is calculated in time $O(n^2)$ [6] with n denotes the maximum number of components that are compared between two individual programs.

We apply the edit distance metric to measure the structural distance between the effective part of programs (*effective distance*) because a difference in effective code may be more directly related to a difference in program behavior (semantic distance). In contrast to a distance metric regarding the full program code (*absolute distance*) this includes some information on program semantics. It is important to realize that effective distance is not part of the absolute distance. Actually, two programs may have a small absolute distance while their effective distance is comparatively large (see Section 5).

Additionally, we regard the sequence of operators (from the effective instructions) only. The sequence corresponding to the example program in Section 5 is $(-, +, /, +, *, -, -, /)$ when starting with the last effective instruction. The distance of effective operator symbols has proven to be sufficiently precise to differentiate between program structures provided that the used operator set is not too small. On the one hand, this is due to the observation that in most cases the modification of an effective instruction changes the effectivity status of at least one instruction. The absolute operator sequence, instead, would not be altered by the exchange of single registers. On the other hand, this metric has been found to guarantee a sufficient correlation with fitness distance (see Section 7.1).

In general, a registration of absolutely every structural difference should not be necessary if we take into account that the correlation between semantic and structural distance is probabilistic. Obviously, less different genotypes are distinguished by our *selective* distance metric that represent the same phenotype (fitness).

Another important motivation for restricting the number of components compared in programs is that the time of distance calculation reduces significantly. Depending on the percentage of non-effective instructions there are k times more elements to compare if one regards the full sequence of operators in programs. Extending the distance measure to registers and constants of instructions, again, results in a factor of 4 maximum. In conclusion, computational cost of the edit distance would increase by a total factor of $(4k)^2$ up to $O(16k^2 \cdot n^2)$.

By using effective mutations we concentrate variation on effective instructions. In this way, step sizes become more closely related to the effective distance metric.

4 Control of Diversity

The effective edit distance between programs is applied for an active control of *genotype diversity*, that is the average *structural* distance between two randomly selected individuals in population. In order to control this distance we introduce the two-level tournament selection shown in Figure 1. On the first level, individuals are selected by fitness. On the second level, the two individuals with *maximum* distance are chosen among *three fitter* individuals, i.e., tournament winners of the first round. While an absolute measure like fitness may be compared between two individuals selection by a relative measure like distance or diversity necessarily requires a minimum of three individuals. In general, two out of n individuals are selected that show the largest sum of distances to the $n-1$ other individuals. While selection pressure on the first level depends on the *size* of fitness tournaments the pressure of diversity selection on the second level is controlled by the *number* of these tournaments. Additionally, a probability parameter controls how often diversity selection takes place.

The number of fitness calculations does not increase with the number of (first-level) tournaments if fitness of all individuals is saved and is updated only after variation. Only diversity selection itself becomes more computationally expensive the more individuals participate in it. Because n selected individuals require $\binom{n}{2}$ distance calculations an efficient distance metric is important here.

The multi-objective selection method prefers individuals that are fit *and* diverse in relation to others. In the two-level selection process fitness selection keeps a higher priority than diversity selection. Selecting individuals only by diversity for a certain probability, instead, does not result in more different directions among *better* solutions in the population. Dittrich *et al.* [5] report on a spontaneous formation of groups when selecting the most distant of three individuals that are represented by single real numbers.

In general, an explicit control of structural diversity increases the average distance of individuals. Graphically, the population spreads wider over the fitness landscape. Thus, there is a lower probability that the evolutionary process gets stuck in a local minimum and more different search directions may be explored in parallel.

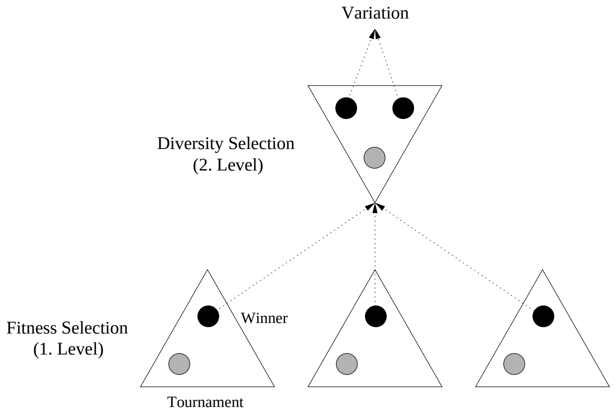

Fig. 1. Two-level tournament selection.

Controlling *phenotype diversity* by a selection for maximum *fitness* distance of individuals has been found less suitable here. Note that program fitness is related to an absolute optimum. Increasing relative fitness distance necessarily increases the diversity of fitness values in population which promotes worse solutions. Moreover, selection by fitness distance has almost no effect on problems that implicate a rather narrow and discrete fitness distribution.

5 Control of Variation Distance

One property of program representations in GP is that already smallest variations on the structural level may affect program behavior heavily. In linear GP these variations especially include the exchange of single registers. Several instructions that precede a varied instruction in a program may become effective or non-effective respectively. In this way, such micro mutations may not only affect the fitness but the flow of data in linear genetic programs. Even if bigger variations of program behavior are less likely with smaller structural variation steps, this effect is rather undesirable.

An *implicit control* of structural variation distance may be realized by imposing respective restrictions on the variation operators [4]. Unfortunately, a variation operator – even if it is operating on the effective code exclusively – can only guarantee for the *absolute* program structure that a certain maximum step size is not exceeded. Variation steps on the *effective* code may still be much bigger though bigger steps appear with a lower probability.

The concern of this contribution is an *explicit control* of the *effective* variation distance. Therefore, the structural distance between parent and offspring is measured explicitly by using the effective distance metric. The variation of a parent program is repeated until its effective distance to the offspring falls below a *maximum threshold*.

In the following extract of a linear program commented instructions are non-effective if we assume that the output is held in register `r[0]` at the end of execution. The program status on the right represents the result of applying a micro mutation to instruction number 8 (from the top). The first operand register `r[3]` is exchanged by register `r[2]`. As a consequence, five preceding (formerly non-effective) instructions become effective which corresponds to an effective mutation distance of five.

```
void gp(r)                                   void gp(r)
  double r[5];                                 double r[5];
{                                            {
   ...                                          ...

// r[4] = r[2] * r[4];                       // r[4] = r[2] * r[4];
   r[4] = r[2] / r[0];                          r[4] = r[2] / r[0];
// r[0] = r[3] - 1;                             r[0] = r[3] - 1;
// r[1] = r[2] * r[4];                          r[1] = r[2] * r[4];
// r[1] = r[0] + r[1];                          r[1] = r[0] + r[1];
// r[0] = r[3] - 5;                             r[0] = r[3] - 5;
// r[2] = pow(r[1], r[0]);                      r[2] = pow(r[1], r[0]);
   r[2] = r[3] - r[4];                          r[2] = r[2] - r[4];     <- mutation point
   r[4] = r[2] - 1;                             r[4] = r[2] - 1;
   r[0] = r[4] * r[3];                          r[0] = r[4] * r[3];
// r[4] = pow(r[0], 2);                      // r[4] = pow(r[0], 2);
// r[1] = r[0] / r[3];                       // r[1] = r[0] / r[3];
   r[3] = r[2] + r[3];                          r[3] = r[2] + r[3];
   r[4] = r[2] / 7;                             r[4] = r[2] / 7;
// r[2] = r[2] * r[4];                       // r[2] = r[2] * r[4];
   r[0] = r[0] + r[4];                          r[0] = r[0] + r[4];
   r[0] = r[0] - r[3];                          r[0] = r[0] - r[3];
}                                            }
```

An alternative metric to the edit distance between effective operator sequences is applicable for controlling step sizes of effective mutations. It simply calculates how many instructions have changed their effectivity status from the mutation point to the beginning of a program. This is exactly the *Hamming distance* between the status flags which needs calculation time $O(n)$ only (with n is the maximum program length here). Even if both metrics calculate similar distances we stick to the edit distance here for consistency reason.

Using an explicit control of the *fitness* distance between parent and offspring, instead, requires an additional fitness calculation after each iterated variation and can become computationally expensive, especially if a larger number of fitness cases is involved. By comparison, a structural distance like edit distance has to be re-calculated only once after each iteration while its computational costs do not directly depend on the number of fitness cases. It is also difficult to find appropriate maximum thresholds for fitness distance because those are usually problem-specific. Finally, it is not sensible to restrict *positive* fitness changes (fitness improvement) at all.

6 Experimental Setup

All techniques discussed above have been tested with three benchmark problems including an approximation, a classification, and a Boolean problem. Table 1 summarizes problem attributes and problem-specific parameter adjustments of our LGP system.

Table 1. Problem-specific parameter settings.

Problem ID	*sinpoly*	*iris*	*8-parity*
Problem type	Approximation	Classification	Boolean function
Problem function	$sin(x) \times x + 5$	real-world data set	even-N-parity (N=8)
Input range	$[-5, 5]$	$[0, 8)$	$\{0, 1\}$
Output range	$[0, 7)$	$\{0, 1, 2\}$	$\{0, 1\}$
Number of inputs	1	4	8
Number of outputs	1	1	1
Number of registers	1+4	4+2	8+0
Number of examples	100	150	256
Fitness function	SSE	CE	SE
Number of generations	500	500	250
Instruction set	$\{+, -, \times, /, x^y\}$	$\{+, -, \times, /, \mathit{if} >, \mathit{if} \leq\}$	{AND, OR, NOT, *if*}
Set of constants	$\{1, .., 9\}$	$\{1, .., 9\}$	$\{0, 1\}$

The first problem is referred to as *sinpoly* in the following and denotes an approximation of the sinus polynomial $sin(x) \times x + 5$ by non-trigonomical functions. Besides the input register – that is identical to the output register here – there are four additional calculation registers used with this problem. This additional program memory becomes important in linear GP, especially if the number of inputs is low by problem definition. Program fitness is the *sum of square errors* (SSE) between the predicted outputs and the example outputs.

The second problem *iris* is a popular classification data set that originates from the *UCI Machine Learning Repository* [2]. The real-world data contains 3 classes of 50 instances each, where each class refers to a type of iris plant. Fitness equals the *classification error* (CE), i.e., the number of wrongly classified inputs. A program output $p(\boldsymbol{i_k})$ is considered as correct for an input vector $\boldsymbol{i_k}$ if the distance to the desired class identifier $o_k \in \{0, 1, 2\}$ is smaller than 0.1, i.e., $|p(\boldsymbol{i_k}) - o_k| < 0.1$.

Finally, we have tested a parity function of dimension eight (*even-8-parity*). This function outputs one if the number of set input bits is even, otherwise the output is zero. Note that the Boolean branch in the instruction set is essential for a high number of successful runs with this problem.

Table 2. General parameter settings.

Parameter	Setting
Population size	2000
Fitness tournament size	4
Maximum program length	200
Initial program length	2–20
Reproduction	100%
Micro mutation	25%
Macro mutation	75%
Instruction deletion	33%
Instruction insertion	67%
Crossover	75%

More general configurations of our linear GP system are given in Table 2. Exactly one genetic operator is selected at a time to vary an individual program. Moreover, either crossover or macro mutations are used as macro (growth) operator, but not in the same run. Macro mutations include two times more insertions than deletions here. This explicit growth tendency of the operator guarantees a sufficient growth of programs. Program length is meassured in number of instructions.

7 Results

7.1 Causality

First of all, we demonstrate experimentally that there is a causal connection between the structural distance and the semantic distance (fitness distance) of linear genetic programs when applying the edit distance metric on sequences of effective instruction operators as defined in Section 3.

The effective variation distance is meassured with both crossover and effective mutations. In both cases, Figure 2 demonstrates a clear positive correlation between program distance and fitness distance, i.e., shorter variation distances on code level induce shorter variation distances on fitness level. The respective distribution of variation distances in Figure 2 confirms this to be true for the vast majority of occurring distances. While, in general, shorter variation distances occur more frequently than longer distances, distribution of crossover distances is wider than the distribution of distances induced by (effective) mutations.

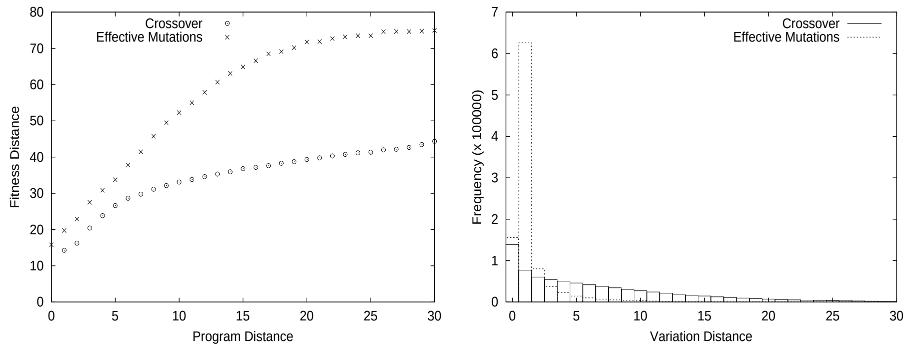

Fig. 2. Relation between program distance and fitness distance (left) and distribution of variation distances (right) for *iris* problem (similar for *sinpoly* and *8-parity*). Average figures over 100 runs.

Furthermore, distance distributions show that almost two thirds of all effective mutations result in distance one. Interestingly, even though macro mutations that insert or delete *full* effective instructions are applied in the majority of cases the effectivity of other (preceding) instructions changes for less than one third only. Obviously, evolution develops effective program structures which are less fragile against stronger variation. We found that the effectivity of an instruction is very often guaranteed by more than one (succeeding) instruction. When crossover is used the proportion of non-effective instructions in a program acts as another implicit mechanism that reduces variation strength.

7.2 Structural Diversity Selection

Table 3 shows average error rates obtained with and without selecting for structural diversity for the three test problems introduced in Section 6. Different selection pressures, i.e., selection probabilities, have been tested with three fitness tournaments. Higher selection pressures are induced by increasing the number of tournaments up to four or eight.

Table 3. Second-level selection for structural diversity with different selection pressures. Selection pressure controlled by selection probability and number of fitness tournaments (T). Average error over 200 runs. Statistical standard error in parenthesis. Percental difference from baseline results.

Variation	Selection		sinpoly		iris		8-parity	
	%	#T	*mean* (*std*)	Δ (%)	*mean* (*std*)	Δ (%)	*mean* (*std*)	Δ (%)
Crossover	0	2	3.01 (0.35)	0	2.11 (0.10)	0	58 (3.4)	0
	50	3	2.89 (0.34)	4	1.42 (0.08)	33	35 (2.4)	40
	100	3	2.77 (0.34)	8	1.17 (0.07)	44	27 (2.2)	53
	100	4	1.96 (0.22)	35	1.09 (0.07)	48	19 (1.8)	67
	100	8	0.69 (0.06)	77	—	—	—	—
Effective Mutations	0	2	0.45 (0.04)	0	0.84 (0.06)	0	15 (1.2)	0
	50	3	0.43 (0.03)	4	0.63 (0.05)	25	12 (1.0)	20
	100	3	0.30 (0.02)	33	0.60 (0.05)	29	10 (1.1)	33
	100	4	0.23 (0.02)	49	0.33 (0.04)	61	7 (0.8)	53
	100	8	0.17 (0.01)	62	—	—	—	—

It is conspicuous that in all three test cases linear GP works significantly better by using (effective) macro mutations instead of crossover. In [4] we have already demonstrated that the linear program representation, in particular, is much more suitable for being developed by mutations, especially if those are directed towards effective instructions. Nonetheless, the experiments with linear crossover show here that diversity selection is not depending on a special type of variation. Moreover, the application of this technique is demonstrated with a population-dependent operator. For each problem and both variation operators performance increases continuously with the influence of diversity selection in Table 3. The highest selection pressure tested for a problem results in a twofold or higher improvement of prediction error. To achieve this, problem *sinpoly* requires a stronger pressure with crossover than the two discrete problems.

Figure 3 illustrates the development of structural diversity during run for different selection pressures. The higher the selection pressure is adjusted the higher is the diversity. Interestingly, the average (effective) program distance does not drop even if diversity selection is not applied. Instead of a premature loss of diversity we observe an inherent increase of structural diversity with our linear GP approach. While diversity increases with crossover until a certain level and stays rather constant then, increase with effective mutations is more linear.

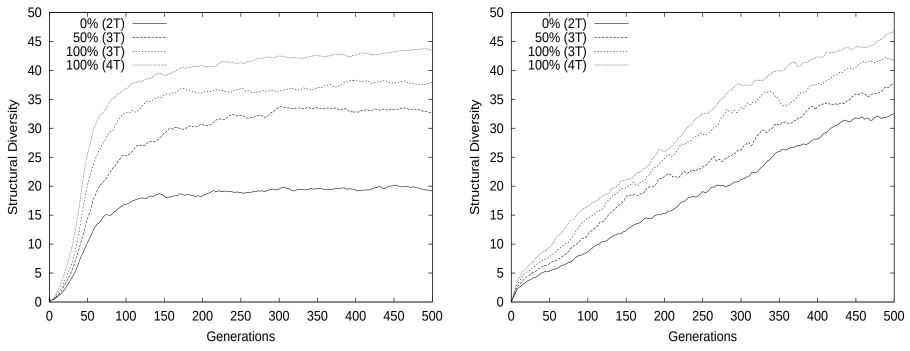

Fig. 3. Diversity levels after diversity selection with different selection pressures. Selection pressure controlled by selection probability and number of fitness tournaments (T). Macro variation by crossover (left) or effective mutations (right) for *iris* problem (similar for *sinpoly* and *8-parity*). Average figures over 100 runs.

Two major reasons can be found to explain this behavior: First, genetic programming is working with a variable-length representation. The longer effective programs develop during a run the bigger effective distances may become. The growth of effective code is more restricted with crossover than with effective mutations because a much higher proportion of non-effective code emerges with this operator – approximately 50–60% in the experiments conducted here. Nevertheless, by the influence of distance selection the average (effective) program length has been found to increase only slightly compared to the average program distance.

Second, both forms of variation, linear crossover and effective mutation, maintain program diversity over a run already implicitly, i.e., without an explicit distance control. For linear crossover the reason might lie in its high variation strength and in the higher amount of non-effective code that contributes to a preservation of (effective) code diversity, too.

When using mutations exclusively a high degree of innovation is introduced continuously into the population. This leads to a higher diversity than it occurs with crossover. The stronger it is selected for diversity, however, the more diversity is gaining ground in crossover runs. Compared to mutations the success of recombination depends more strongly on the composition of the genetic material in the population. The more different the recombined solutions are the higher is the expected innovativity of their offsprings.

7.3 Control of Effective Mutation Distance

As motivated in Section 5 we are interested in controlling the *effective* distance between parent and offspring. We restrict ourselves to the mutation-based approach here. In particular, we want to find out whether solution quality may be further improved by an explicit reduction of effective mutation distances. Therefore, a program is mutated repeatedly until its distance to the offspring falls below a maximum threshold. Each time a mutation is not accepted its effect on the program is reversed while the choice of the mutation point is free in every iteration.

The applied effective distance metric regards operators as smallest distance units (see Section 3). This corresponds to (effective) macro mutations which operate on instruction level (*macro level*), even if the effective distance may also be altered by micro mutations that operate *below* instruction level (*micro level*). In order to guarantee a sufficient code growth, macro mutations are applied more frequently than micro mutations here. As a result, most steps by effective mutations are larger than zero (about 80% in Figure 2) and measuring the distance between *full* effective programs does not promise a much higher precision. This is another reason, besides the arguments given in Section 3, why (effective) operator sequences represent a sufficient basis for distance calculation between linear genetic programs.

Table 4. Maximum restriction of effective mutation distance. Average error over 200 runs. Statistical standard error in parenthesis. Percental difference from baseline results.

Variation	Maximum	sinpoly		iris		8-parity	
	Distance	*mean* (*std*)	Δ (%)	*mean* (*std*)	Δ (%)	*mean* (*std*)	Δ (%)
Effective	—	0.46 (0.06)	0	0.90 (0.06)	0	16 (1.2)	0
Mutations	10	0.35 (0.04)	24	0.72 (0.06)	20	13 (1.2)	19
	5	0.33 (0.04)	28	0.74 (0.06)	18	12 (1.2)	25
	2	0.28 (0.03)	39	0.68 (0.05)	24	11 (1.1)	31
	1	0.26 (0.03)	42	0.54 (0.05)	40	9 (0.9)	44

Table 4 compares average prediction errors for different maximum limits of mutation distance. The maximum possible distance equals the maximum program length and imposes no restrictions. For all three benchmark problems best results are obtained with the smallest maximum distance (1). This is all the more interesting if we consider that a restriction of variation distance always implies a restriction in variation freedom.

As we can see in Table 5 the average number of iterations during run increases only slightly if the maximum threshold is lowered. Not even one and a half iterations are necessary, on average, for the smallest distance and the maximum number of iterations (10 here) has hardly ever been exceeded. Both aspects together with the results from Table 4 emphasize that freedom of variation is not restricted significantly and that computational costs of this distance control are not expensive.

The results found here further correspond to the distribution of mutation distances in Figure 2 where only about 20–30% of all measured step sizes are larger than one. Obviously, larger disruptions of effective code as demonstrated with

Table 5. Average number of iterations until a maximum mutation distance is met.

Variation	Maximum	Iterations		
	Distance	sinpoly	iris	8-parity
Effective	—	1.00	1.00	1.00
Mutations	10	1.02	1.02	1.02
	5	1.06	1.05	1.05
	2	1.18	1.12	1.12
	1	1.37	1.18	1.20

the example program in Section 5 occur less likely. Effective parts of programs rather emerge to be quite robust against bigger effective mutations steps.

8 Future Work and Conclusion

A two-level tournament selection may also be used for implementing a complexity control. Compared to a weighted complexity term in the fitness function (*parsimony pressure*) [10], fitness selection is less influenced by a *complexity selection* on the second level and finding an appropriate weighting of objectives is not required. Moreover, the separation of linear genetic programs in effective and non-effective code offers the possibility for a *selective* complexity selection. That means it may be selected for smallest effective length or smallest non-effective length specifically.

We introduced an active control of diversity in form of a two-level selection process. By increasing the structural distance between programs, performance improved significantly for three different benchmark problems. Measuring structural differences specifically between effective subcomponents of linear genetic programs was found sufficiently precise to demonstrate causality.

We also restricted the mutation distance on level of effective code. This turned out to be most successful with the smallest maximum step size while the number of necessary repetitions of a mutation was small. In general, mutation distances on effective linear programs were found much smaller than it might be expected.

Acknowledgements. This research was supported by the Deutsche Forschungsgemeinschaft (DFG), collaborative research center SFB 531, project B2.

References

1. W. Banzhaf, P. Nordin, R. Keller and F. Francone, *Genetic Programming – An Introduction. On the Automatic Evolution of Computer Programs and its Application.* dpunkt/Morgan Kaufmann, Heidelberg/San Francisco, 1998.
2. C.L. Blake and C.J. Merz, *UCI Repository of Machine Learning Databases* [`http://www.ics.uci.edu/~mlearn/MLRepository.html`]. University of California, Department of Information and Computer Science.

3. M. Brameier and W. Banzhaf, *A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining.* IEEE Transactions on Evolutionary Computation, vol. 5(1), pp. 17–26, 2001.
4. M. Brameier and W. Banzhaf, *Effective Linear Program Induction.* Collaborative Research Center SFB 531, Computational Intelligence, Technical Report No. CI-108/01, University of Dortmund, 2001.
5. P. Dittrich, F. Liljeros, A. Soulier, and W. Banzhaf, *Spontaneous Group Formation in the Seceder Model.* Physical Review Letters, vol. 84, pp. 3205–3208, 2000.
6. D. Gusfield, *Algorithms on Strings, Trees and Sequences.* Cambridge University Press, 1997.
7. C. Igel and K. Chellapilla, *Investigating the Influence of Depth and Degree of Genotypic Change on Fitness in Genetic Programming.* In W. Banzhaf et al. (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1061–1068, MIT Press, Cambridge, 1999.
8. E.D. de Jong, R.A. Watson, and J.B. Pollack, *Reducing Bloat and Promoting Diversity using Multi-Objective Methods.* In L. Spector et al. (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 11–18, MIT Press, Cambridge, 2001.
9. R. Keller and W. Banzhaf, *Explicit Maintenance of Genetic Diversity on Genospaces*, Internal Report, University of Dortmund, 1995.
10. J.R. Koza, *Genetic Programming.* MIT Press, Cambridge, MA, 1992.
11. P. Nordin, *A Compiling Genetic Programming System that Directly Manipulates the Machine-Code.* In K.E. Kinnear (ed.) *Advances in Genetic Programming*, 311–331, MIT Press, Cambridge, MA, 1994.
12. I. Rechenberg, *Evolutionsstrategie '94.* Frommann-Holzboog, 1994.
13. U.-M. O'Reilly, *Using a Distance Metric on Genetic Programs to Understand Genetic Operators.* In J.R. Koza (ed.), *Late Breaking Papers at the Genetic Programming '97 Conference*, Standford University, 1997.
14. J.P. Rosca and D.H. Ballard, *Causality in Genetic Programming.* In L.J. Eshelmann (ed.), *Proceedings of the Sixth International Conference on Genetic Algorithms*, pp. 256–263, Morgan Kaufmann, San Francisco, 1995
15. D. Sankoff and J.B. Kruskal (eds.), *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, 1983.
16. R. Tanese, *Distributed Genetic Algorithms.* In J.D. Schaffer (ed.) *Proceedings of the Third International Conference on Genetic Algorithms*, 434–439, Morgan Kaufmann, San Mateo, CA, 1989.