Effective Linear Genetic Programming

Markus Brameier   Wolfgang Banzhaf

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

Abstract

Different variants of genetic operators are introduced and compared for linear genetic programming including program induction without crossover. Variation strength of crossover and mutations is controlled based on the genetic code. Effectivity of genetic operations improves on code level and on fitness level. Thereby algorithms for creating code efficient solutions are presented.

1 Introduction

In genetic programming conventionally two genetic operators, e.g. crossover and mutation, are used for varying program solutions during an evolutionary process. This is valid for tree-based as well as for linear program representations. While crossover is responsible for larger variation steps by exchanging subprograms of arbitrary length between individuals, mutations take effect on single atomic units of the program structure. In linear GP subprograms denote sequences of instructions that operate on register variables and constants [1, 3]. In linear programs exchanging a single register can have a significant influence on the program flow. This is in contrast to single point mutations in tree-based GP. Moreover, influence of tree crossover is smoothed by choosing crossover points with a higher probability the deeper they are located in the tree hierarchy. In this way, the average size of the exchanged subtrees is reduced implicitly. We will show in this contribution that linear crossover with an unrestricted exchange of subprograms is not very recommendable. An alternative method to smooth the variation strengh of the linear crossover operator is homologous crossover [5].

Motivated by the observation that variation step sizes become already quite big with mutations in linear GP we compare different configurations of the linear crossover operator with the exclusive use of macro mutations that are insertions and deletions of single instructions. Chellapilla [4] has shown for tree-based GP that program induction by using macro mutations can be competitive with subtree crossover. But he still allows several genetic operators to be applied at a time. Instead, we recalculate fitness after every operation in order to gain a clearer understanding and further restrict variation strength.

The linear genetic code of a specific program solution can be divided into two classes of instructions, effective and noneffective. While only the effective instructions may have an influence on the program result the latter class may be designated as "introns". Because the proportion of noneffective code is usually significantly high many genetic operations are neutral in terms of a fitness change. In [3] the noneffective code has been removed before fitness calculation. By doing so, linear GP is accelerated significantly. In this paper we increase the efficiency of genetic operations by concentrating them on the effective part of the code. Different variants of effective mutation and effective crossover operators are introduced for the linear GP approach.

The major goals of our study are summarized as follows:

- Reduction of variation step sizes in linear genetic programming.
- Increase in the rate of effective genetic operations.
- Reduction of neutral operations.

Using two benchmark problems we will demonstrate that both lower variation strength and more effective operations produce better performance in approximation as well as in generalization. Further, program solutions become smaller in size whereby the effect of program bloat is reduced significantly for linear GP.

2 Basics on Linear GP

Linear genetic programming [1] applies a linear representation of individual programs. Instead of tree-based GP expressions of a functional programming language (like LISP) but programs of an imperative language (like C or machine code) are evolved. In the linear GP system used for our experiments [3] an individual program is represented by a variable-length sequence of simple C instructions. All instructions operate on one or two indexed variables (*registers*) r or constants c from predefined sets and assign the result to a destination register, e.g. $r_i = r_j * c$. The operation set used for the experiments here includes *addition, subtraction, multiplication, division* and *exponentiation.*

3 Effective Linear Programs

In linear genetic programs two types of code can be distinguished. Depending on their position in the program and the register they manipulate there are *effective* and *noneffective* instructions.

Definition 1 (*effective instruction*): An instruction of a linear genetic program is *effective* at its position if it influences the output(s) of the program for at least one possible input situation. *Noneffective* instructions, respectively, are without any influence on the calculation of the output(s) for *all* possible inputs.

The program structure in linear GP allows noneffective code to be identified efficiently. Algorithm 1 achieves this in linear runtime $O(n)$, where n is the maximum length of a program. An extended version of the algorithm has been introduced in [3] to extract effective code from individuals once before fitness is calculated. By saving the execution of the noneffective code during program interpretation the evolutionary process is accelerated significantly.

Algorithm 1 (*Detection of all effective instructions in a linear genetic program*):

1. Let the set of registers $R_{eff} := \{ r \mid r \text{ is output register } \}$ always contain all registers which have an influence on the program output at the current position.
   Start from the last instruction and move backwards.
2. Mark the next operation with *destination* register $r \in R$.
   If such an instruction is not found, $\rightarrow 4$.
3. Insert the *operand* register(s) of the newly marked instructions in R_{eff} if not already contained. $\rightarrow 2$.
4. Stop. All marked instructions are *effective*, i.e. have an influence on the program output at position i.

An excerpt of a linear genetic program is given below in C notation. All instructions shown with an exclamation mark are effective if register r[0] holds the final output.

```
void gp(r)
  double r[9];
{
  ...

  r[2] = r[8] * r[5];
! r[2] = ppow(r[0], r[6]);
! r[1] = pdiv(r[1], 2);
! r[4] = ppow(r[5], r[1]);
! r[6] = r[3] - r[2];
  r[7] = ppow(r[6], 5);
  r[0] = pdiv(r[3], r[4]);
  r[1] = r[7] * 1;
! r[2] = ppow(r[4], r[6]);
  r[1] = r[2] + r[2];
  r[7] = r[2] * 6;
! r[0] = r[8] + 7;
  r[4] = pdiv(r[2], 5);
  r[7] = r[1] - 8;
! r[7] = r[2] + 2;
  r[1] = r[7] - r[7];
! r[5] = pdiv(r[6], r[7]);
  r[1] = r[3] * 6;
  r[6] = r[1] - r[0];
  r[1] = ppow(r[0], r[7]);
! r[0] = r[0] + r[5];
}
```

In tree-based GP noneffective code is not an inherent part of the program structure. The proportion of subtrees that do not alter the program output strongly depends on the composition of function set and terminal set. These subtrees may be either executable or non-executable, e.g. a branch holding a nonsatisfiable condition. Because the noneffective code depends on the semantics of the program its detection is more difficult and not all noneffective parts can be found [6].

In linear GP the (*structural*) noneffective code is independent of the applied instruction operators and is detected completely by Algorithm 1. In order to restrict the rate of *semantical* introns in the effective code and, thus, to keep the effective length of programs small instructions can be choosen with a minimum tendency for creating these introns.

4 Variation step sizes in linear GP

On the *structural* level each position of a linear genetic program can be manipulated with the same degree of freedom and without changing the rest of the program code. For tree-based GP, instead, it is more difficult to delete or insert a group of nodes (or subtree) at an arbitrary position. Usually complete subtrees have to be removed along with the operation to satisfy the constraints of the tree structure.

On the *semantical* level the situation becomes more complicated for linear GP. As already mentioned in the introduction it is possible that exchanging a single register (index) in an instruction effects data flow and fitness heavily. Several instructions that precede the mutated instruction in the program may become effective or noneffective respectively. Consequently, semantical variation strength is already quite high with single point mutations.

Another characteristic of linear genetic programs complicates the realization of smoother variation operators, i.e. operators with a smaller variation step size. In program trees crossover and mutation points can be expected to be the more influencial to program semantics the closer they are to the root of the tree. In a linear GP program, instead, each position of an instruction may have a similar influence on the program behaviour: A linear genetic program can be transformed into a tree representation by a successive replacement of variables starting with the last effective instruction. It is obvious that such a tree would grow exponentially with effective program length and could become extremely large. These trees usually contain many identical subtrees close to the leafs. For that reason the effect of an instruction may not only be large at the end of a linear program (because close to the tree root) but also at the beginning (because multiply represented in the tree).

Both arguments motivate restrictions of the freedom in variation for standard genetic operators used in linear GP. Especially the crossover operator exchanges segments of instructions between two individuals independently of how much genetic material is altered. One simple way of reducing the effect of linear crossover is a limitation of the segment length. Another alternative is to use macro mutations instead of crossover to vary program length. Both approaches reduce variation step sizes in linear GP on the structural (code) level and are one topic of this study. This practice relies on the general assumption in genetic programming that smaller structural variations of programs lead to smaller semantical variations with a high probabilty.

We restrict *macro mutations* to insertions or deletions of single instructions since they guarantee a minimum effect on program structure. Changing the position of an instruction (swapping) or the replacement of an instructions, for instance, are more destructive since they include both a deletion and an insertion at the same time. Only after the maximum program length has been reached insertions are allowed to replace other instructions.

In tree-based GP the effect of macro mutations on the program structure is more difficult to control and usually includes deletions or insertions of complete (random) subtrees [4]. Smaller variation steps (structural and semantical) are only possible if deeper nodes of a tree are affected.

5 Effective Genetic Operators

According to the definition of effective code from Section 3 we define effective operations as follows:

Definition 2 (*effective operation*): A genetic operation applied to a genetic program is called *effective* if it modifies the effective code.

Note that even if the effective code (see Definition 1) is altered by an operation the predictions of the program for a considered set of fitness cases can be the same. An effective operation is merely meant to bring about a *structural* change of the effective program code. There is not always a change of the program semantics (fitness) guaranteed, too, which is due to (*semantical*) intron code [3]. In general, decreasing the number of noneffective operations is expected to reduce the rate of neutral operations as well:

Definition 3 (*neutral operation*): A genetic operation is *neutral* if it is does *not* change the fitness of a program.

Table 1 lists the different types of mutation operators and crossover operators—including their effective opponents—that are investigated and compared in this paper. It is guaranteed for each operation that there is a (structural) variation of the program code. Otherwise the operation is repeated. Especially identical replacements of code elements, i.e. registers, constants or instructions, are explicitly avoided during mutations. Further, only one genetic operator is applied at a time, i.e. before the next fitness calculation is performed, to keep evolutionary step sizes of code variation as small as possible. Micro mutation of constants are smoothed by choosing a different value within a standard deviation from the current one.

Genetic Operator	Definition
Standard crossover	Exchange of arbitrary long sequences of instructions between two programs.
Limited crossover	Exchange of program segments of limited maximum length.
Effective crossover	Crossover between two *effective programs*.
Macro mutation	Insertion or deletion of a random instruction.
Micro mutation	Exchange of an operand or operator of an instruction.
Effective mutation	An *effective instructions* is selected for mutation.

Table 1: (Effective) genetic operators for linear GP.

Note that *either* crossover *or* macro mutations are applied in the same experiment because one interest of this work is a comparsion of linear GP with and without using recombination. Micro mutations, instead, are used in addition to both growth operators. With these two basic approaches of (linear) GP we further compare effective variants that apply corresponding effective operators.

There are two possibilities to increase the number of effective operations. Either the proportion of noneffective code is reduced or operations are concentrated on the effective part more intensively. *Effective programs* are programs whose noneffective code has been removed completely (using Algorithm 1). This happens only to all new individuals in the population directly after being created with the *effective crossover* approach. In [2] Blickle proposed to remove redundant code parts before tree crossover. His intention was to control code growth on the one hand and to increase the effectivity of crossover operations on the other hand. The latter was achieved by simply reducing the probability that a crossover point falls upon a redundant subtree and makes the operation useless for that program.

Two different approaches are considered for *effective mutations*. One variant uses macro (and micro) mutations to operate on effective instructions exclusively but leaves the noneffective code untouched. This is strongly motivated by the assumption that mutations on noneffective instructions may be more likely invariant according to the fitness than modifications of the effective code. Another variant of effective macro mutations allows (single) noneffective instructions to be deleted from the code. Note that both variants allow the replacement of noneffective instructions if the maximum program length has been reached. Otherwise, the size of effective code (*effective length*) could not increase any further. In a third approach *all* emerging noneffective instructions might be deleted directly from the population as done with effective crossover. But this has been found too much restrictive for program growth, at least for the parameter configuration choosen here (see Section 6). Only with maximum insertion rate (100%), i.e. without explicit deletions, competitive results are possible then.

The deletion of an instruction in general is uncomplicated. The deletion of a noneffective instruction does not even require any recalculation of the fitness (like all noneffective operations). The insertion of an effective instruction, instead, requires the application of Algorithm 2 to construct an instruction that is effective for a given program position. Explicit insertions of noneffective instructions (introns) are avoided, of course.

Algorithm 2 (*Insertion of an effective instruction at a position i*):

1. Determine all effective registers at position i by terminating Algorithm 1 after that position has been reached. Set R_{eff} holds the effective registers then.

2. Choose the *destination* register for the new instruction from R_{eff}.
   The choice of the *operand* register(s) is free.

3. Shift all instructions from position i (inclusively) to the end of program one position downwards and insert the new instruction at position i.

6 Test Problems and System Configuration

The performance of the different genetic operators introduced above and their influence on solution complexity has been compared using two benchmark problems. Note that similar results have been found with a couple of other regression problems that are not documented here.

The first problem is represented by a two-dimensional *mexican hat* function (see Equation 1). Figure 1 shows a three-dimensional plot of the function visualizing the surface that has to be approximated.

$$mexicanhat(x,y) = \left(1 - \frac{x^2}{4} - \frac{y^2}{4}\right) * e^{\left(-\frac{x^2}{8} - \frac{y^2}{8}\right)} \tag{1}$$

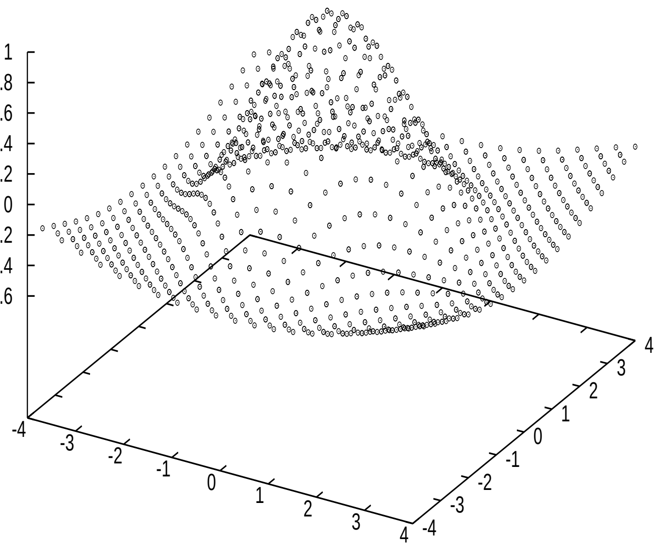

Figure 1: *Mexican hat* function.

The second test problem, *two points*, computes the square root of the scalar product of two three-dimensional vectors p and q (see Equation 2).

$$twopoints(p_x, p_y, p_z, q_x, q_y, q_z) = \sqrt{p_x * q_x + p_y * q_y + p_z * q_z} \tag{2}$$

Tables 2 summarizes complexity attributes of the data samples selected for each problem. These are input and output dimension, input range as well as number of randomly selected examples for training set, validation set and test set. It is important for the performance of linear GP to provide enough registers for calculations, especially if input dimension is low. Therefore, the number of (*calculation*) registers—additional to the registers that hold the input data—is an important parameter. In general, the number of registers decides on the number of program paths that can be calculated in parallel. If it is not sufficient there are too many conflicts by overwriting of register information within programs.

General configurations of our linear GP system are given in Table 3. The instruction set that has been selected for approximating both problem functions is not complete, i.e. it does not include the *square root* or the *exponential* function (e^x) explicitly, but is sufficient to develope the optimum solution.

Problem	Inputs	Input range	Outputs	Samples		
				Training	Validation	Testing
mexican hat	2	$[-4, 4]$	1	400	400	400
two points	6	$[0, 10)$	1	200	200	200

Table 2: Problem dimensions.

Parameter	Setting
Number of generations	1000
Population size	1000
Maximum program size	200 instructions
Initial maximum size	20 instructions
Instruction set	$\{+, -, \times, /, x^y\}$
Initial set of integer constants	{1,..,9}
Micro mutations	25%
Macro mutations	75%
Deletions	33%
Insertions	66%
Crossover	75%

Table 3: General parameter settings.

In order to guarantee that the rate of new individuals is the same with crossover and macro mutations, always two tournament winners are recombined or both of the two parents undergo macro mutation. Tournament selection is applied with the minimum of two participants per tournament.

Macro mutations include two times more insertions than deletions here. This *explicit growth* tendency of the operator has proven advantageous with both problem tasks. In general, inserting and deleting single instructions with the same probability, instead, may result in a growth of programs which is too slow for producing good results.

7 Experiments and Results

In this Section the different variants of linear GP that have been introduced in Section 5—application of crossover or macro mutations on one hand and their effective equivalents on the other hand—are compared according to prediction performance, generalization and code complexity. All experimental results presented are average figures of 60 test runs.

Tables 4 and 5 document prediction results for the *two points* and the *mexican hat* problem. *Fitness* (*training error*) is calculated as the sum of square errors between predicted and desired program outputs. The respective error on the validation set (*validation error*) is checked during evolution among the best individuals while the best validating solution is retained and tested on unknown data at the end of a run (*test error*).

In our crossover experiments (cross) the maximum length of the exchanged code segments has been varied through three different orders of magnitude (2, 20 and 200 instructions). Note, that the actual segment lengths are selected randomly from a range of one to current program length if this is smaller than the maximum segment length. Amazingly, the

Operator	Segment Length	Fitness	Validation Error	Testing Error	Effective Registers	Neutral Operations
cross	**200**	192.1	276.5	274.7	4.0	218
	20	171.0	259.6	268.1	4.3	257
	2	100.8	138.0	157.4	4.5	298
effcross	**200**	352.5	405.5	422.6	3.8	137
	20	324.2	367.0	409.4	4.0	134
	2	395.0	454.5	450.1	2.0	150

Operator	Calculation Registers	Fitness	Validation Error	Testing Error	Effective Registers	Neutral Operations
mut	0	157.0	261.4	263.9	3.3	396
	3	95.5	165.6	195.2	4.5	419
	6	94.0	168.8	189.2	5.1	476
	12	86.4	147.0	194.6	5.8	554
	24	100.0	167.6	201.9	6.8	637
effmut	0	90.6	127.7	143.0	3.8	73
	3	66.3	92.7	95.5	5.2	66
	6	53.6	75.0	72.1	6.4	68
	12	44.9	70.7	76.7	8.4	95
	24	45.3	76.1	78.4	11.6	114
effmut2	**3**	76.5	114.0	123.9	5.2	56

Table 4: Prediction quality for the *two points* problem Bold printed results correspond to Figures 2 and 3.

smaller the exchanged code parts are the better results become for training and generalization. This is a strong hint that the unlimited standard crossover operator (maximum segment length equals maximum program length 200) is varying genetic programs too heavily.

If the emerging noneffective code is completely removed directly after every operation as with the effective crossover approach (effcross), the prediction quality of linear genetic programs decreases significantly for the *two points* problem. Variations generated by the operator on the (effective) code are most destructive then. This experiment confirms that noneffective code is needed to control the variation strength when using recombination by crossover. On the one hand, noneffective instructions act as (*structural*) introns that protect the effective code from being disrupted too heavily. Although because of the maximum limitation of program length (200 instructions) this implicit control of crossover step size does not prove sufficient. On the other hand, there is a significant loss of genetic material when removing the noneffective code. This leads to a premature loss of diversity and a slower growth of (effective) program size—especially at the beginning of a run where segment lengths are comparatively small (see below). Noneffective (intron) code, instead, preserves code diversity in the population and allows longer and better effective code to emerge with crossover.

With effective crossover evolution tries to compensate the absence of noneffective code by increasing (effective) program length—especially for the *mexican hat* problem (see Figures

Operator	Segment Length	Fitness	Validation Error	Testing Error	Effective Registers	Neutral Operations
cross	**200**	15.46	17.68	17.73	2.2	200
	20	12.95	15.50	15.44	2.3	238
	2	3.44	4.63	4.47	2.7	235
effcross	**200**	11.61	13.46	13.34	2.6	102
	20	12.63	15.86	15.80	2.7	107
	2	17.09	17.86	17.66	2.2	131

Operator	Calculation Registers	Fitness	Validation Error	Testing Error	Effective Registers	Neutral Operations
mut	0	12.00	15.72	15.73	1.4	361
	2	5.77	9.94	10.25	2.5	408
	4	2.64	5.64	5.43	3.4	430
	8	1.33	3.84	3.66	4.7	480
	16	1.45	2.78	2.46	6.4	549
effmut	0	4.58	5.72	5.59	1.8	99
	2	1.35	1.92	1.69	2.8	75
	4	1.08	1.93	1.68	3.8	92
	8	0.90	1.63	1.38	5.4	114
	16	0.65	1.21	0.99	8.1	143
effmut2	**2**	1.59	2.25	2.05	2.72	88

Table 5: Prediction quality for the *mexican hat* problem Bold printed results correspond to Figures 4 and 5.

3 and 5). But this intron formation within the effective code (*semantical introns*) does not always prove a sufficent protection mechanism. With normal crossover these introns are generally (see Section 3) more difficult to create and to maintain. Hence their emergence may be suppressed in the presence of a sufficient amount of noneffective code. In this way, the size of the noneffective code may reduce the size of the effective code, at least if crossover is used.

In contrast to noneffective crossover, prediction results get worse with smaller segment lengths. The reason is that the growth of programs (not documented) is too restricted to compensate for the permanent loss of genetic material caused by the radical deletion of noneffective code.

Exchanging one or two instructions during crossover only is a very special kind of recombination, all the more if you take into account that in most cases the number of possible instructions is comparatively small. This motivates the study of macro mutations instead of crossover that insert or delete a single (random) instruction only. Note that the linear crossover operator guarantees an exchange, i.e. deletion *and* insertion, of at least one instruction.

Tables 4 and 5, lower part, compare fitness and generalization performance when using mutations exclusively (mut) for different numbers of additional (calculation) registers. For both benchmark problems macro mutations performed better than standard crossover if the same number of calculation registers is provided—three registers with *two points* and two registers with *mexican hat*. Moreover, results with mutation have been found to lie within the same order of magnitude as "crossover" configuations with very restricted segment lengths. As a conclusion, crossover should be definitely restricted in influence. One simple but efficient way is the reduction of segment lengths.

It may be added that the number of neutral operations (from 1000 genetic operations per generation) reduces when applying genetic operators that concentrate on the effective code. Obviously, avoiding noneffective operations leads to more operations that are relevant to fitness. Since, in general, neutral operations are more likely with mutations than with crossover this effect is more significant with effective mutations than with effective crossover. With effective mutations (effmut) the number of neutral operations reduces significantly compared to mutations that are not restricted to the effective code. Though larger code parts are varied by crossover there can still be more neutral operations with effective crossover than with effective mutations. The reason is simply that the exchanged crossover segments are not necessarily effective in other programs as well. With mutations instead this may be guaranteed explicitly (see Section 5).

The clearest improvement in results is achieved in our experiments through effective mutations (effmut). Prediction performance with effective mutations still improves or, at most, level out with a very high amount of calculation registers compared to the number of input registers. Interestingly, this effect has been observed with normal mutations, too, for the *mexican hat* problem. But results may get worser here again beyond an optimum number of registers as found with the *two points* problem. The main reason is that effective insertions (see Section 5) are much more independent from the input dimension since registers are *selected* effectively here.

The average number of *effective registers* that are *used* at a particular program position correlates with the number of program paths that are calculated in parallel in a linear GP program. It depends not only on the problem but directly on the number of registers provided. Especially for effective mutations this number increases according to the number of calculation registers (see Table 4).

The variant of effective mutations applied in our effmut experiments allows deletions of noneffective instructions (see Section 5). By doing so, the rate of noneffective code reduces to less than one percent. For comparison reason we added an experiment with effective mutations (effmut2) which are restricted to effective instructions only. Although noneffective code is not touched by this variant it maintains a much lower rate of noneffective code than standard macro mutations (mut, see Figures 3 and 5). In case of effmut the noneffective code may function as a small protection mechanism that controls the influence of user-defined deletion rate (33% here). This might explain the (slightly) lower prediction performance of effmut2 solutions (see Tables above). Just as the little difference of effmut and effmut2 solutions in effective size this could result from negative side-effects by reactivation of noneffective (intron) code as well.

Figures 3 and 5 document that effective lengths are around 50% smaller if solutions are developed with effective mutations (effmut) than with normal macro mutations (mut). This might originate from the problem that it is more difficult for evolution to preserve shorter effective code if operators tend to produce a lot of noneffective instructions. Since only effective code is executed in our linear GP system processing time is accelerated significantly by using effective mutations. Regarding the respective prediction results from Tables 4 and 5 the smaller effective solutions show much better performance in training

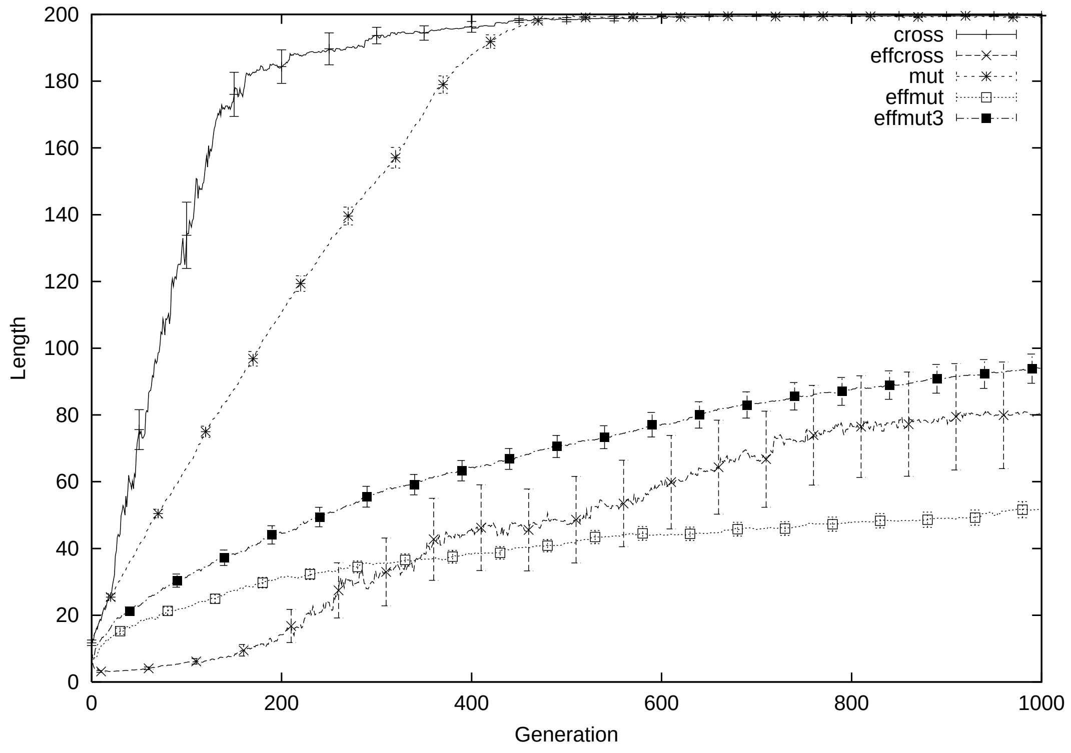

Figure 2: Development of program lengths with *two points* problem. Average taken over 60 runs. Error bars indicate standard error.

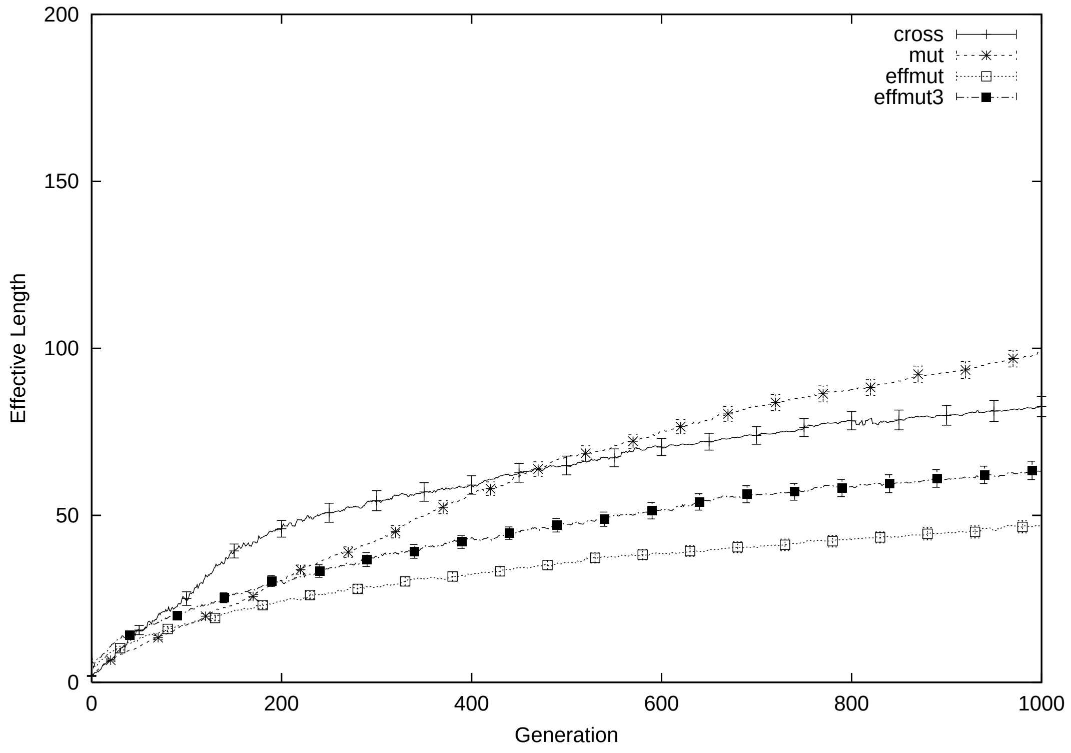

Figure 3: Development of effective lengths with *two points* problem. For effcross effective length equals absolute length (see Figure 2).

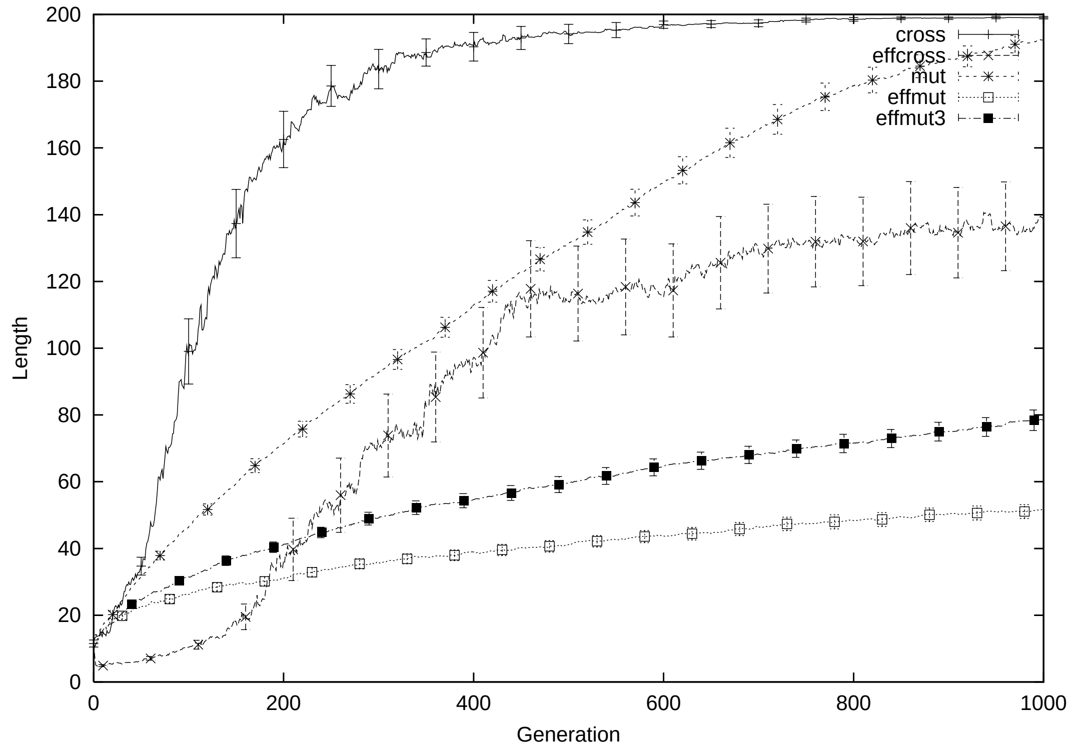

Figure 4: Development of program lengths with *mexican hat* problem. Average taken over 60 runs. Error bars indicate standard error.

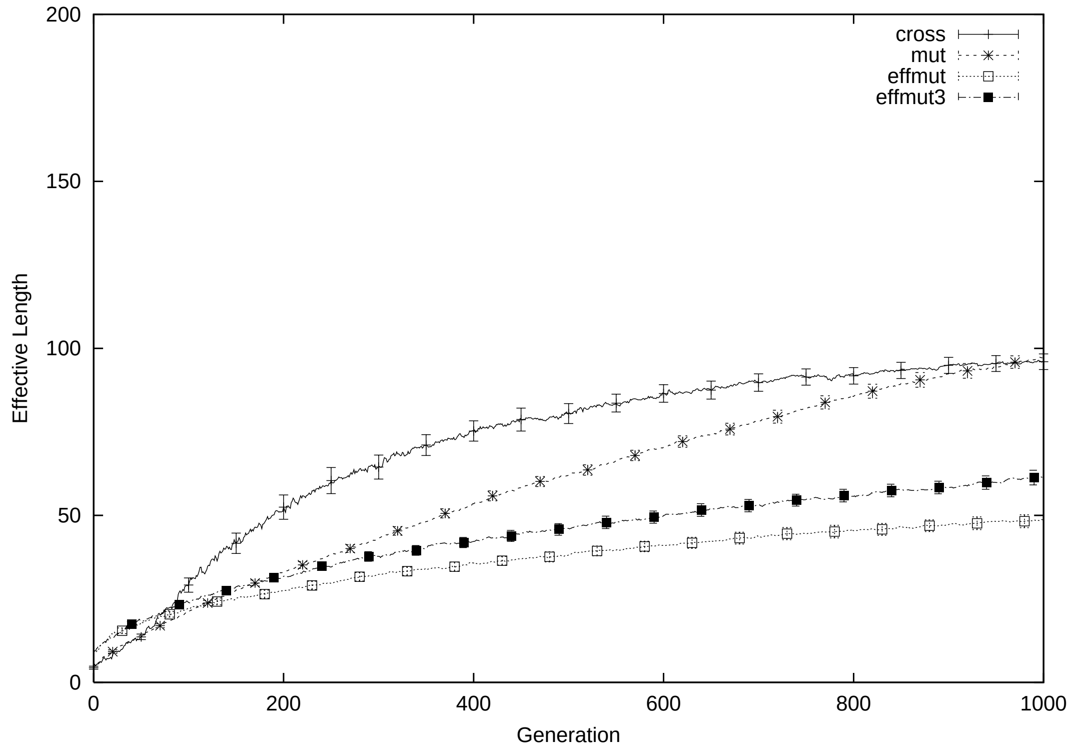

Figure 5: Development of effective lengths with *mexican hat* problem. For effcross effective length equals absolute length (see Figure 4).

and generalization here.

Finally, we compare the development of effective length with the absolute lengths in Figures 2 and 4. With unlimited standard crossover the absolute length approaches the maximum size limit of 200 instructions within the first 300 generations. With noneffective macro mutations this effect is extentuated by reaching the maximum later. Obviously, program growth is slower with mutations. Because noneffective code does not influence fitness directly there is no selection pressure on this part of a program. If crossover is applied the protection function of the noneffective code becomes an additional drive of program growth. Obviously, this so called bloat effect is caused by an explosive increase of the noneffective code part in linear GP. This has been found for the growth of tree programs as well [6].

Effective operations in general reduce the rate of noneffective code and, thus, the bloat effect significantly. This is true for effective mutations even without allowing deletions of noneffective instructions (effmut2). In this way, effective macro mutations offer an implicit control of code growth for linear genetic programming. In contrast to crossover effective macro mutations do not require noneffective code necessarily for inducing efficient solutions, neither for controlling variation strength nor for preserving code diversity.

Besides a slower increase in program length figures for mutations are smoother than length figures for crossover. Both observations reflect that variation step sizes are smaller on the code level when using mutations.

8 Conclusion

Several possibilities to decrease variation steps of linear genetic operators have been introduced and have proven to be successful in improving fitness and generalization quality. Program induction by using macro mutations exclusively has not only turned out to be competitive with linear crossover but developed significantly better solutions for two benchmark problems. Moreover, effective macro mutations offer a successful control of code growth in linear genetic programming while evolving more compact solutions.

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] T. Blickle, *Evolving Compact Solutions in Genetic Programming: A Case Study.* In H-M Voigt, W. Ebeling, I. Rechenberg, H-P Schwefel, editors, Parallel Problem Solving from Nature - PPSN IV, pp. 564–573, LNCS 1141, Springer, Berlin, Germany, 1996.

[3] M. Brameier and W. Banzhaf, *A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining. IEEE Transactions on Evolutionary Computation*, in press.

[4] K. Chellapilla, *Evolving Computer Programs Without Subtree Crossover. IEEE Transactions on Evolutionary Computation*, vol. 1, no. 3, 1997.

[5] P. Nordin, W. Banzhaf and F. Francone, *Efficient Evolution of Machine Code for CISC Architectures using Blocks and Homologous Crossover. Advances in Genetic Programming III*, pp. 275–299, MIT Press, Cambridge, MA, 1999

[6] T. Soule, J. A. Foster, and J. Dickinson, *Code growth in genetic programming.* In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, Genetic Programming 1996: Proceedings of the First Annual Conference, pp. 215–223, MIT Press, Cambridge, MA, 1996.