1 Genetic Programming and Its Application in Machining Technology

Wolfgang Banzhaf[1], Markus Brameier[1], Marc Stautner[2], and Klaus Weinert[2]

[1] University of Dortmund, Department of Computer Science, Informatik XI, 44221 Dortmund, Germany
{banzhaf, brameier}@Ls11.cs.uni-dortmund.de
[2] University of Dortmund, Faculty of Mechanical Engineering, Institute of Machining Technology, 44221 Dortmund, Germany
{stautner, weinert}@isf.mb.uni-dortmund.de

Summary. Genetic programming (GP) denotes a variant of evolutionary algorithms that breeds solutions to problems in the form of computer programs. In recent years genetic programming has become increasingly important for real-world applications, including engineering tasks in particular. This contribution integrates both further development of the GP paradigm and its applications to challenging problems in machining technology. Different variants of program representations are investigated. While problem-independent methods are introduced for a linear representation, problem-specific adaptations are conducted with the traditional tree structure.

1.1 Introduction

In the first part of this chapter we focus on advanced concepts of linear GP. Linear genetic programming (LGP) denotes a variant of GP, where programs of an imperative programming language or a machine language are evolved. We analyze data flow within linear genetic programs and exploit structural aspects of the linear representation for detecting noneffective code. Extracting this code from programs before fitness calculation leads to a significant acceleration of runtime. The information about whether an instruction is effective or not is further used for designing efficient variation operators. Here we present a variant of linear GP that is based exclusively on mutations. Program solutions developed with this approach have not only been found to be more successful but less complex in size. Furthermore, a structural distance metric is defined that reveals causal connections between changes of the linear genotype and changes of the phenotype (fitness). Using this distance information allows variation step size and diversity to be controlled explicitly on the level of effective code.

The second part deals with the application to problems of machining technology. Here modeling general correlations out of data given only by observation is a difficult task. A lot of knowledge about the examined process is needed

to extract the correct coherences out of the given data. Especially when this process has to be done by hand. To support the scientist in this problem computational intelligence (CI) methods, here symbolic regression through genetic programming, are a well known possibility to help.

1.2 Linear Genetic Programming

Genetic programming [2,16] applies an evolutionary algorithm that subjects computer programs to evolution. In contrast to traditional GP which uses expressions of a functional programming language, linear GP evolves programs of an imperative language. While in the former case the program structure is a tree, in the latter case the representation is linear, which derives from the sequence of imperative instructions such a genetic program is composed of.

Originally, linear genetic programming was introduced with a binary machine language so that the genetic programs can be executed directly without passing a time-consuming interpretation step first [21]. Apart from this speed advantage, we investigate more general characteristics of linear representations in this chapter. One principle difference compared to tree representations is that unused code parts occur and remain within linear genetic programs (see Section 1.2.1).

In our linear GP approach [4], an individual program is represented by a variable-length sequence of simple C instructions. The instruction set may include operations such as $r_i = r_j + c$, function calls $r_i = f(r_j)$, or branches $if(r_j > r_k)$ that may skip the succeeding non-branch instruction. All instructions operate on one or two indexed variables (registers) r_j or constants c from predefined sets and all instructions, but branches assign the result to a destination register r_i. The individual LGP approach has been extended to the evolution of program teams in [6].

1.2.1 Effective and Noneffective Instructions

In linear genetic programs two types of code can be distinguished on the structural level, effective and noneffective instructions. An instruction is defined as effective at its program position if it manipulates an effective register. A register, in turn, is effective at a certain position if a manipulation of its content can effect the behavior, i.e., the output(s), of the program. Otherwise, the register as well as the manipulating instruction are noneffective for that position. Such noneffective instructions do not contribute to the data flow in a program and are referred to as structural introns or data flow introns.

In the following excerpt of a linear genetic program, only the instructions that are marked with an exclamation point can have an influence on the final output, which is held in register `r[0]` here.

Example 1: (*Linear genetic program*)

```
void gp(r)
  double r[4];
{
   ...

   r[3] = r[1] - 3;
   r[1] = r[2] * r[1];
!  r[3] = r[1] / r[0];
   r[0] = r[1] - 1;
   r[1] = r[2] * r[0];
   r[1] = r[0] * r[1];
!  r[0] = r[2] + r[2];
   r[2] = pow(r[1], r[0]);
!  r[2] = r[0] + r[3];
!  r[0] = r[3] - 1;
!  r[1] = r[2] - r[0];
!  r[3] = pow(r[0], 2);
!  r[2] = r[2] + r[1];
   r[0] = r[1] + 9;
   r[0] = r[1] / r[3];
!  r[0] = r[2] * r[2];
!  r[2] = r[1] * r[3];
!  r[0] = r[0] + r[2];
}
```

The structural noneffective code in linear individuals is not directly dependent on the set of instruction operators. However, the more registers that are provided, the easier the creation of this code will become for evolution. If only one register is available, this type of code cannot occur at all, and finding a solution is very difficult.

Semantic (or operational) intron instructions manipulate effective registers without affecting program semantics. In order to restrict the rate of semantic introns and thus, to keep the structural effective length of programs small, an instruction set can be chosen with a minimal tendency to create these introns. Only if the creation of structural introns is easier than the creation of semantic introns, semantic noneffective code can be expected to occur less frequently.

1.2.2 A GP Algorithm

Algorithm 1 shows the specific evolutionary algorithm used in our LGP approach. In such a steady-state EA, newly created individuals replace existing individuals in the same population, while typically, tournament selection is applied.

Algorithm 1: (*LGP algorithm*)

1. Initialize a population of individual programs randomly.
2. Select $2 \times n$ individuals from the population without replacement.
3. Perform two fitness tournaments of size n.
4. Reproduce the winners of the two tournaments by replacing the two losers with temporary copies of the winners.
5. Modify the two winners by one or more variation operators, including mutation and recombination, with certain probabilities.
6. Evaluate the fitness of the offspring.
7. If the currently best-fit individual is replaced by one of the offspring then validate the new best program.
8. Repeat steps 2–8 until the maximum number of generations has been reached.
9. Test the generalization performance of the program with minimum validation error.
10. Both the best program during training and the best program during validation define the output of the algorithm.

The fitness (training error) of an individual program p is often computed by an error function, e.g., the sum of errors between the predicted and the desired program outputs for a given set of n training examples (also called fitness cases):

$$fitness(p) = \sum_{k=1}^{n} |p(i_k) - o_k| \tag{1.1}$$

Better fitness means smaller error and the minimum fitness is zero then. The data domain from which the input-output examples (i_k, o_k) are drawn defines the problem that should be solved or approximated by GP.

The generalization ability of LGP solutions is checked during training by calculating the validation error of the currently best-fit program. This requires the use of unknown data (validation data), which is sampled from the same data space as the training data. Finally, among all the best individuals emerging over a run the one with minimum validation error (point of best generalization) is tested on a test data set again once after training is over.

1.3 Removal of Noneffective Code

The most time-consuming step in GP is the evaluation of individuals. Usually this step includes multiple executions of a genetic program. In [21] execution

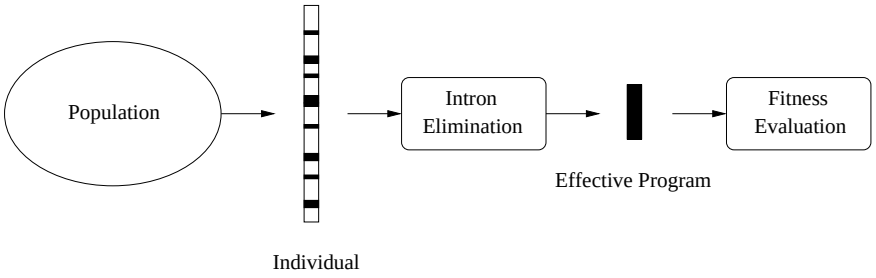


Fig. 1.1. Intron elimination in LGP. Only effective code (black) is executed.

time has been reduced by evolving binary machine programs that do not require interpretation. In this section we introduce another possibility to speed up linear GP.

The imperative program structure in linear GP permits noneffective instructions to be identified efficiently. This in turn allows the effective code to be extracted from a program and to be copied to a temporary program buffer once before the fitness of the program is calculated (see Figure 1.1). By only executing the effective program when testing each fitness case, evaluation can be accelerated significantly [4]. It is important to realize that the representation of the individuals in the population is not affected by this action. In this way, no potential genetic material gets lost.

Algorithm 2 detects all structural introns in a linear genetic program. Note that whether or not a branch instruction is an intron only depends on the status of the operation that directly follows. Copying all marked instructions at the end forms the effective program.

Algorithm 2: (*Detection of noneffective instructions*)

1. Let the set R_{eff} always contain all registers that are effective at the current program position.
   $R_{eff} := \{ r_i \mid r_i \text{ is output register} \}$
   Start at the last program instruction and move backwards.
2. Mark the next operation with destination register $r_i \in R$.
   If such an instruction is not found, $\rightarrow 5$.
3. If the operation directly follows a branch or a sequence of branches, mark these instructions too, otherwise remove r_i from R_{eff}.
4. Insert the operand register(s) of new marked instructions in R_{eff} if not already contained. $\rightarrow 2$.
5. Stop. All unmarked instructions are introns.

The algorithm needs linear runtime $O(n)$ at worst, with n being the maximum length of the linear genetic program. Actually, detecting and removing

the noneffective code from a program only requires about the same time as calculating one fitness case.

By omitting the execution of noneffective instructions during program interpretation a large amount of computation time can be saved. A good estimate of the overall acceleration in runtime is the factor

$$\alpha_{acc} = \frac{1}{1 - p_{intron}} \tag{1.2}$$

with p_{intron} denotes the average percentage of intron code in a genetic program and $1 - p_{intron}$ the respective percentage of effective code. In [4] an average intron rate of about 80 percent is documented for several classification problems, which corresponds to an average decrease in runtime by a factor 5. In general, the proportion of effective and noneffective instructions in linear programs depends on both the problem and the configuration of the LGP system, including the provided program components (see Section 1.2) and the choice of variation operators (see Section 1.5) in particular.

1.4 Graph Interpretation

The imperative representation of a linear GP program can be transformed into a semantically equivalent functional program graph. The following Algorithm 3 achieves this for an effective linear program. Thus, this method requires the deletion of intron instructions by using Algorithm 2, first. The directed structure of the resulting graph better reflects functional connections and data flow in linear genetic programs. The graph is contiguous for two reasons: first, by not transforming structural intron instructions and, second, by assuming that there is only one output register. Finally, the graph is acyclic if instructions, such as loops or backward jumps do not occur in the program. Instructions, such as conditional branches result in graph structures that are not always visited during execution. Both rather special cases of programming concepts, iterations and branches, shall be excluded from the following discussion for simplicity.

Algorithm 3: (*Transformation of an effective linear program into a graph*)

1. Start at the last instruction in the program. Its destination register r_{dest} is the output register of the program and labels the start node of the graph (sink node at this stage).
2. Go to the sink node in the graph that holds the destination register r_{dest} of the current instruction as a label.
3. Assign the instruction operator to this node.
4. For each operand register r_{op} (and constant) of the current instruction repeat steps 5–7:

5. If there is no sink node with label r_{op}, create it.
6. Connect r_{dest} and r_{op} by a directed edge. (r_{dest} becomes inner node.)
7. If not all operations are commutative label this edge with 1 (2) if r_{op} is the first (second) operand.
8. If the end of program is reached, $\rightarrow$ 10.
9. Go to the next preceding instruction in program. $\rightarrow$ 2.
10. Stop. Delete all register labels from inner nodes.

Each program instruction may be interpreted as a small subtree of depth one. In the effective program graph resulting from Algorithm 3, each inner node represents an operator and has as many outgoing edges as there are operands in the corresponding instruction, i.e., one or two.[1]

Sink nodes, i.e., nodes without any outgoing edges, are labeled with register identifiers or constants. Sink nodes that represent a constant are only created once during the calculation of the graph. Only sink nodes that represent a variable (register) are replaced regularly by operator nodes in the course of the algorithm. These are the only points at which the graph may grow. Since loops are not considered the only successors of sink nodes may become other sink nodes or new nodes. After each iteration of the algorithm all nonconstant sink nodes correspond exactly to the effective registers at the current program position. Because the number of effective registers is limited by the total number of registers, of course, the number of sink nodes and the maximum width of the graph respectively are limited as well. As a result, the program graph is supposed to grow in depth. The depth is restricted by the length of the imperative program because each imperative instruction corresponds to exactly one inner node in the graph. For those reasons the graph structure may be referred to as linear, like its imperative equivalent.

The actual width of a program graph indicates the number of parallel calculation paths in a linear genetic program. It can be determined by the maximum number of registers that are effective at a program position. It is important for the performance of linear GP to provide enough registers for calculations (calculation registers), in addition to the registers that hold the input data. This is especially true if input dimension is rather low. If the number of registers is not sufficient there are too many conflicts through overwriting of register information during calculation. The more registers are provided, however, the more independent program paths may develop.

It also follows from the above observations that the runtime of Algorithm 3 is $O(k \cdot n)$ with n is the number of effective instructions and k is the number of registers. For problems with a rather low input dimension, i.e., with a small number of input registers, runtime is almost linear in n.

The program listed in Example 1 corresponds exactly to the graph in Figure 1.2 after applying Algorithm 3. Both, the imperative representation

[1] The maximum number of operands in an instruction is two.

and the functional representation consist of effective code here that is free from unused instructions or unvisited graph components, respectively. This is valid if we assume that the output of the imperative program is stored in variable `r[0]` at the end of execution. At the beginning of program execution the inputs are held in variables `r[0]` to `r[3]`. In the corresponding graph representation the inputs denote variable sink nodes, besides the sink nodes that hold constants.

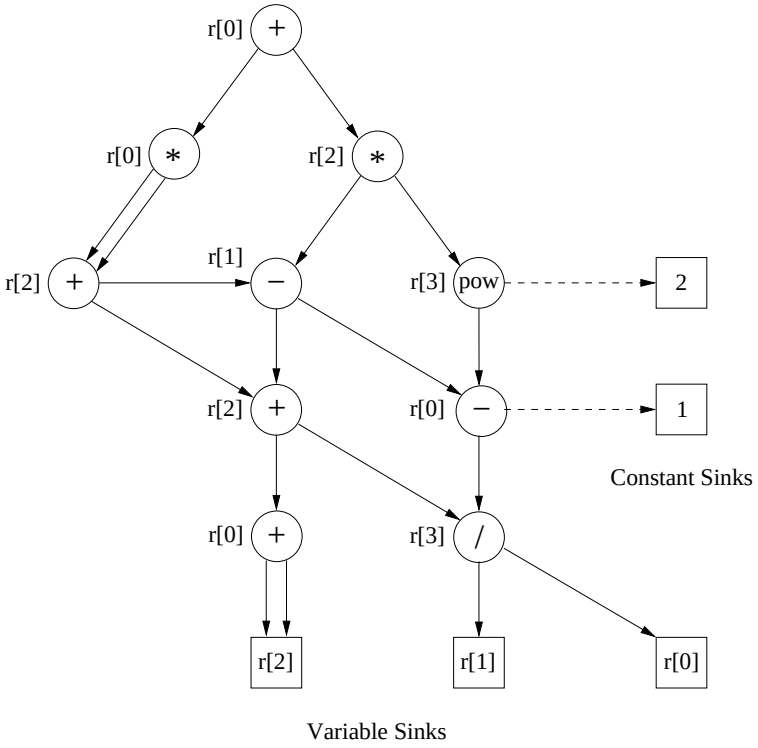


Fig. 1.2. Graph equivalent of the effective linear program code in Example 1. Operator nodes are labeled with the destination registers of the corresponding instructions (see Algorithm 3). Output register r[0] marks the start node. Outgoing edges are not labeled here but printed in the right order.

1.5 Linear Genetic Operators

Traditionally, recombination is applied in genetic programming for varying the lengths of programs. In this section we introduce an efficient variant of linear GP that works exclusively with mutations. We will demonstrate that

linear programs are manipulated more precisely by using mutations only. This will also be motivated by means of the functional interpretation discussed in Section 1.4.

The linear crossover operator exchanges two arbitrarily long subsequences 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 subsequences. Macro-mutations denote deletions or insertions of single full instructions here. Both crossover and macro-mutations operate on instruction level and control program growth (macro-variations). Micro-mutations by comparison, randomly replace instruction components that comprise single operators, registers, or constants.

1.5.1 Variation Step Sizes

In genetic programming changing a small program component may lead to almost arbitrary changes in program behavior. Variation operators that induce smaller step sizes with a high probability allow a more precise approximation to locally optimal solutions. This relies on the assumption that, on average, smaller structural variations of genetic programs result in smaller semantic variations (see also Section 1.6).

In tree programs, crossover and mutation points can be expected to be the more influential the closer they are to the root. Instead, in a linear program, each position of an instruction may have a more similar influence on program semantics. Recall that the linear graph representation is restricted in width through the number of registers provided (see Section 1.4).

Moreover, in tree-based GP, crossover only affects a single point in data flow, which is the root of the exchanged subtree. With linear crossover the contents of many effective registers may change, i.e., several points in data flow are modified simultaneously. The reason lies in the rather narrow graph structure of linear genetic programs. This graph is disrupted easily when applying linear crossover on the imperative structure. As a result, crossover step sizes may become quite large, on average. However, it has to be considered that, to a certain degree, the influence of linear crossover on the effective code is reduced implicitly by a high rate of structural intron code that emerges with this operator.

A more explicit way of reducing the effect of linear crossover is to limit the length of the exchanged instruction segments. We experienced that rather small maximum segment lengths produce the best results [5].

The above reasons suggest the use of macro-mutations instead of crossover for macro-variations in linear GP. Mutations guarantee a sufficient variation strength and freedom of variation in linear GP for the following reasons. First, in linear GP already single micro-mutations that exchange a register index in an instruction may change the data flow within a linear program. This is true because several instructions that precede the mutated instruction may become effective or noneffective respectively (see Section 1.6). On

the graph level a single edge is redirected from an effective subgraph to a former noncontiguous component. In tree-based GP micro-mutations usually manipulate the contents of single nodes but do not change edges of the tree structure.

Second, the linear program/graph structure can be manipulated with a high degree of freedom. In tree programs, by comparison, it is rather difficult to delete or insert a group of nodes at an arbitrary position. Complete subtrees might be removed along with the operation to satisfy the constraints of the tree structure [10]. This is why the tree structure is less suitable to be varied by smaller macro-mutations since modification of upper program parts necessarily involve bigger parts of code. In linear GP, depending substructures do not get lost when deleting or inserting an instruction but remain within the linear representation as structural noneffective code, i.e., unvisited graph components.

1.5.2 Effective Mutations

According to the definition of effective code in Section 1.2, let an effective variation denote a genetic operation that modifies the effective code of a genetic program. Note that even if the effective code is altered, the predictions of the program for a considered set of fitness cases might be the same. An effective variation is merely meant to bring about a structural change of the effective program code. There is no change of program semantics (fitness) guaranteed which is due to semantic intron code. In general, decreasing the number of noneffective variations is expected to reduce the rate of neutral variations. A genetic operation is neutral if it does not change the fitness of a program. Note that by definition, the term noneffective is meant to be related to the code level only.

We consider two different approaches to effective mutations. One variant (effmut2) uses macro- and micro-mutations to operate on effective instructions exclusively but leaves the noneffective code untouched. This is motivated by the assumption that mutations of noneffective instructions may be more likely invariant in terms of a fitness change than mutations of effective code. The other variant (effmut) allows single noneffective instructions to be deleted. In order to guarantee that the effective code is altered, an effective insertion may directly follow such intron deletions. A third approach might be to delete all emerging noneffective instructions directly after the variation.[2]

The deletion of an instruction in general is not complicated. If an instruction is inserted, its destination register is chosen in such a way that the instruction is effective at the corresponding program position. By doing so, insertions of noneffective instructions (introns) are avoided explicitly. Like the effective code, effective registers can be identified efficiently in linear runtime $O(n)$ [5].

[2] This has been found too restrictive for program growth because of substantial loss of genetic material.

1.5.3 Prediction Performance

In the next two sections we will compare the performance of linear genetic operators and their influence on solution complexity using two regression problems. The first problem is represented by the two-dimensional mexican hat function that is to be approximated:

$$f_{mexicanhat}(x,y) = \left(1 - \frac{x^2}{4} - \frac{y^2}{4}\right) \times e^{\left(-\frac{x^2}{8} - \frac{y^2}{8}\right)} \tag{1.3}$$

The second test problem, named two points, computed the square root of the scalar product of two three-dimensional vectors p and q:

$$f_{twopoints}(p_1, p_2, p_3, q_1, q_2, q_3) = \sqrt{p_1 \times q_1 + p_2 \times q_2 + p_3 \times q_3} \tag{1.4}$$

Tables 1.1 and 1.2 compare fitness and generalization performance for different forms of macro-variation: crossover (cross), free macro-mutations (mut) and two variants of effective macro-mutations (see above). In the same run micro-mutations are applied together with one of the four macro operators with a probability of 25 percent. Only one genetic operator, however, is applied at a time to vary a certain individual program.

First, we can see that prediction errors reduce together with the variation step sizes when using macro-mutations instead of crossover. Interestingly, an additional significant improvement can be observed with effective mutations. One explanation is that effective mutations, by definition, reduce the number of neutral variations significantly compared to mutations that are not restricted to the effective code. In this way, evolution is allowed to progress faster within the same period of time, i.e., number of generations.

Table 1.1. Mexican hat problem: Average prediction error over 60 runs.

Variation	Training Error	Validation Error	Test Error	Neutral Variations
cross	15.46	17.68	17.73	200
mut	5.77	9.94	10.25	408
effmut	1.35	1.92	1.69	75
effmut2	1.59	2.25	2.05	88

In the effmut2 experiments effective mutations are restricted to effective instructions. Although noneffective code is not touched by this variant, it maintains a much lower intron rate than standard macro-mutations (mut) as can be seen from Figures 1.3–1.6. Instead, the other variant of effective mutations (effmut) allows deletions of noneffective instructions (see Section 1.5.2), which reduces the rate of noneffective code further. The intron code may

Table 1.2. Two points problem: Average prediction error over 60 runs.

Variation	Training Error	Validation Error	Test Error	Neutral Variations
cross	192.1	276.5	274.7	218
mut	95.5	165.6	195.2	419
effmut	66.3	92.7	95.5	66
effmut2	76.5	114.0	123.9	56

function as a protection mechanism that compensates (reduces) the influence of the user-defined deletion rate (33 percent here) on the effective code. This could explain the slightly better prediction performance compared to effmut2 solutions. By applying insertions two times more frequently than deletions, a sufficient code growth is guaranteed, especially with effective macro-mutations.

In general, when using macro-mutations, worse performance may also result from negative effects, in particular, larger mutation step sizes, by reactivation of intron code. Note that noneffective code hardly ever emerges with the effmut variant.

1.5.4 Control of Code Growth

Figures 1.4 and 1.6 document that effective lengths are around 50 percent smaller if solutions are developed with effective mutations (effmut) than with normal macro-mutations (mut). Most probably it is more difficult for evolution to preserve shorter effective code in the presence of many noneffective instructions. In general, a reduction of effective program size always means an acceleration of processing time too because only effective code is executed in our linear GP system (see Section 1.3).

For unlimited standard crossover the absolute length approaches the maximum size limit of 200 instructions within the first 300 generations in Figures 1.3 and 1.5. With free mutations this effect is extenuated by reaching the maximum later. Obviously, program growth is slower by using mutations only. Nevertheless, both variation forms induce a fast increase of the absolute code length. This effect is referred to as bloat and is observed in tree-based GP too [26]. If we compare the development of absolute lengths with the effective lengths in Figures 1.6 and 1.4 it becomes obvious that the bloat effect is mostly related to a growth of the structural noneffective code in linear GP. Because noneffective code does not influence fitness directly, there is no selection pressure on this part of a program. Moreover, the rate of noneffective code works as an implicit control of crossover step sizes. If crossover is used this protection function forms an additional drive for code growth.

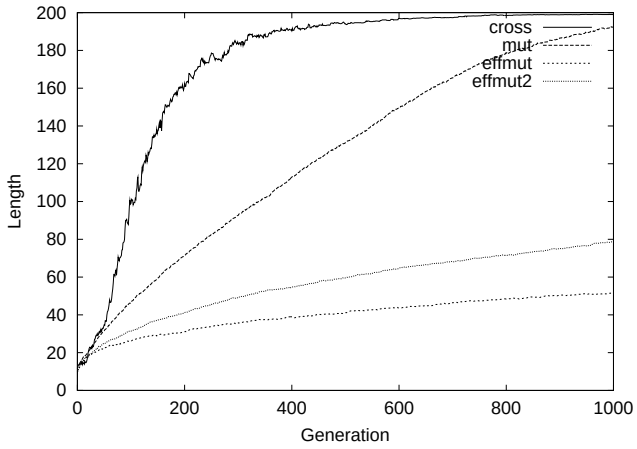


Fig. 1.3. Mexican hat problem: Development of absolute program length for different forms of variation. Average figures over 60 runs.

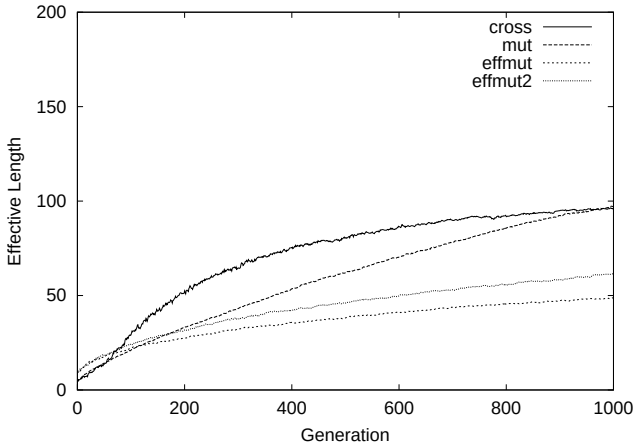


Fig. 1.4. Mexican hat problem: Development of effective program length for different forms of variation. Average figures over 60 runs.

Effective mutations reduce the intron code and thus, the bloat effect significantly, even without allowing deletions of noneffective instructions explicitly (effmut2). In this way, effective mutations incorporate an implicit control of code growth in linear genetic programming. In contrast to crossover, macromutations do not require noneffective code for inducing efficient solutions, neither for controlling variation strength nor for preserving code diversity.

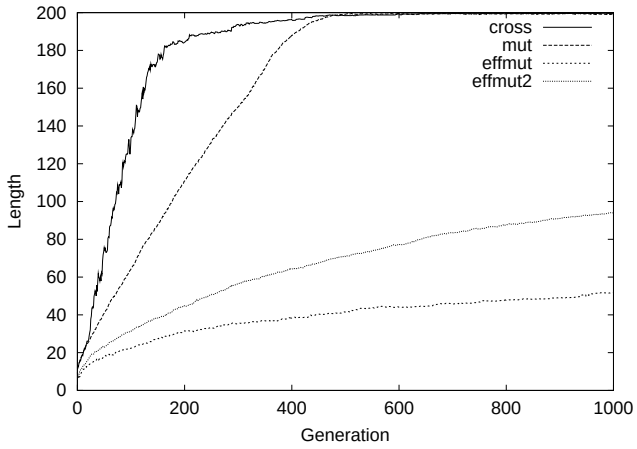


Fig. 1.5. Two points problem: Development of absolute program length for different forms of variation. Average figures over 60 runs.

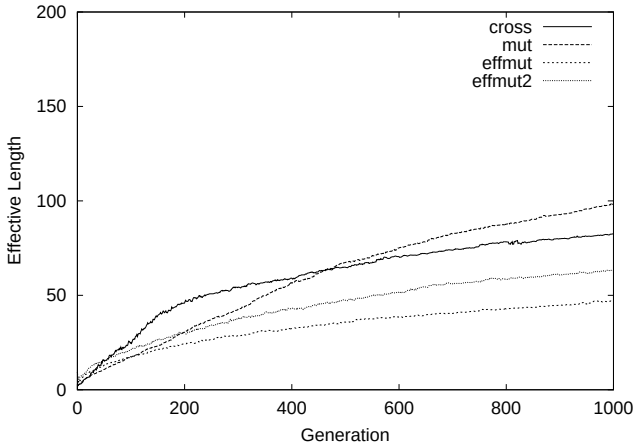


Fig. 1.6. Two points problem: Development of effective program length for different forms of variation. Average figures over 60 runs.

1.6 Control of Variation Step Size

An implicit control of structural variation distance may be realized by imposing respective restrictions on the variation operators. This has been demonstrated above by using macro-mutations instead of recombination in order to vary program length. Moreover, by concentrating variations on the effective code, structural variations become more closely related to semantic variations. Recall that only the degree of variation on the effective code decides the difference in fitness. Unfortunately, a variation operator, even if it is op-

erating on the effective code exclusively, can only guarantee for the absolute program structure that a certain maximum variation step size is not exceeded. Variation steps on the effective code instead may still be much bigger, even though this will happen with a lower probability.

The following example represents the result of applying a micro-mutation to the program example from Section 1.2. In the instruction on line 9 the operand register `r[3]` has been exchanged by register `r[2]`. As a consequence, five preceding, formerly noneffective, instructions become effective now, which corresponds to an effective distance of five (see below).

Example 2: (*Mutated linear program*)

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

   r[3] = r[1] - 3;
!  r[1] = r[2] * r[1];
!  r[3] = r[2] / r[1];
!  r[0] = r[1] - 1;
!  r[1] = r[2] * r[0];
!  r[1] = r[0] * r[1];
!  r[0] = r[2] + r[2];
!  r[2] = pow(r[1], r[0]);
!  r[2] = r[0] + r[2];     <- Effective mutation point
!  r[0] = r[3] - 1;
!  r[1] = r[2] - r[0];
!  r[3] = pow(r[0], 2);
!  r[2] = r[2] + r[1];
   r[0] = r[1] + 9;
   r[0] = r[1] / r[3];
!  r[0] = r[2] * r[2];
!  r[2] = r[1] * r[3];
!  r[0] = r[0] + r[2];
}
```

In order to avoid such bigger variation steps on program structure, we propose an explicit control that repeats a variation until the structural distance between parent and offspring falls below a maximum threshold [7,8].[3] In particular, we want to find out whether a further reduction of effective mutation step size may still improve results. This requires mutation distances to be measured explicitly on the effective code. In the following section we will identify substructures of linear genetic programs that are sufficient to be distinguished by a structural distance metric.

Using an explicit control of semantic distance (fitness distance) between parent and offspring instead would require an additional fitness calculation

[3] Reference [7] is an extended version of [8].

for each repeated variation and can become computationally expensive, especially if a larger number of fitness cases is involved. A structural distance has to be recalculated 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 they are not problem specific. Finally, it is not sensible to restrict fitness improvements at all.

1.6.1 Structural Program Distance

The string edit distance [12] measures the distance between two arbitrarily long character strings by counting the number of basic operations, including insertion and exchange of single elements, that are necessary to transform one string into an other. We apply this distance metric to determine the structural distance between the effective part of parent and offspring (effective variation distance) because a difference in effective code may be more directly related to a difference in program behavior. It is important to realize that the effective distance cannot be part of the absolute distance. Actually, two programs may have a small absolute distance while their effective distance is comparatively large, as has just been demonstrated at the example program above. On the other hand, two equally effective programs might differ significantly in their noneffective code.

For an efficient distance calculation we concentrate on representative substructures of linear programs and regard the sequence of operators from the effective instructions. For instance, the order of effective operators of the genetic program from Section 1.2 is

$$(+, *, *, +, pow, -, -, +, +, /)$$

when starting with the last instruction. The distance of effective operator symbols has been found sufficiently precise to differentiate between program structures, provided that the used operator set is not too small. This is due to the fact that in most cases the modification of an effective instruction changes the effectivity status of (at least) one instruction (see Section 1.6.3). Note that in contrast to the effective distance the absolute operator sequence would not be altered by the exchange of single registers.

Effective mutations work closely with the effective distance metric. As defined in Section 1.5, macro-mutations operate on full instruction level, while micro-mutations vary smaller components within instructions. In order to guarantee a sufficient growth of programs, however, the higher number of variations is performed on macro level, i.e., comprises macro-mutations. Since, in this way, the average step size is not further reducible from operator side, measuring the distance between full effective programs does not necessarily promise a higher precision. This is another reason why operator sequences represent a sufficient basis for distance calculation between linear

genetic programs. Besides, 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 (see Section 1.6.3).

1.6.2 Prediction Performance

As a first benchmark problem we test the above notions with the iris data set that contains popular real-world data from the UCI Machine Learning Repository [3]. Fitness is the classification error, i.e., the number of wrongly classified inputs. The second test problem is a parity function of dimension eight (even-8-parity). This function outputs 1 if the number of set input bits is even, otherwise the output is 0.

Table 1.3. Restriction of effective mutation distance. Average error over 200 runs. Statistical standard error in parenthesis. Percentage difference from baseline results.

Variation	Maximum	iris		even-8-parity	
	Distance	*mean* (*std*)	Δ (%)	*mean* (*std*)	Δ (%)
effmut	—	0.90 (0.06)	0	16 (1.2)	0
	10	0.72 (0.06)	20	13 (1.2)	19
	5	0.74 (0.06)	18	12 (1.2)	25
	2	0.68 (0.05)	24	11 (1.1)	31
	1	0.54 (0.05)	40	9 (0.9)	44

Table 1.4. Average number of iterated mutations until a maximum distance is met.

Variation	Maximum	Iterations	
	Distance	iris	even-8-parity
effmut	—	1.00	1.00
	10	1.02	1.02
	5	1.05	1.05
	2	1.12	1.12
	1	1.18	1.20

Table 1.3 compares average prediction errors for different maximum mutation distances. The maximum possible distance equals the maximum program length (200 instructions) and imposes no restrictions. For both problems, the best results are obtained with the smallest effective distance (one). This is all

the more interesting if we consider that a restriction of variation distance always implies a restriction in variation freedom too. More specifically, certain modifications might not be executed at certain program positions because too many other instructions would be affected.

As we can learn from Table 1.4, the average number of iterations until a maximum effective distance is met increases only slightly if the maximum threshold is lowered. On average, only ≈ 1.2 iterations are necessary with the smallest threshold, and the maximum number of iterations (10 here) has hardly ever been exceeded. Both aspects emphasize that freedom of variation is restricted only slightly here.

1.6.3 Distance Distribution and Correlation

The results in Table 1.4 correspond to the distribution of effective mutation distances in Figure 1.7, where only about 20 percent of all measured step sizes are larger than one. Obviously, larger disruptions of effective code, as demonstrated with the example program in Section 1.6, occur less likely. Effective programs emerge to be quite robust against larger mutation steps because their survival probability is higher in this way. In particular, this is achieved by the effectivity of an instruction that may depend on more than one succeeding instruction in program. Because the rate of noneffective instructions that emerge with effective mutations is comparatively low, most effective mutation distances result from deactivation of effective instructions.

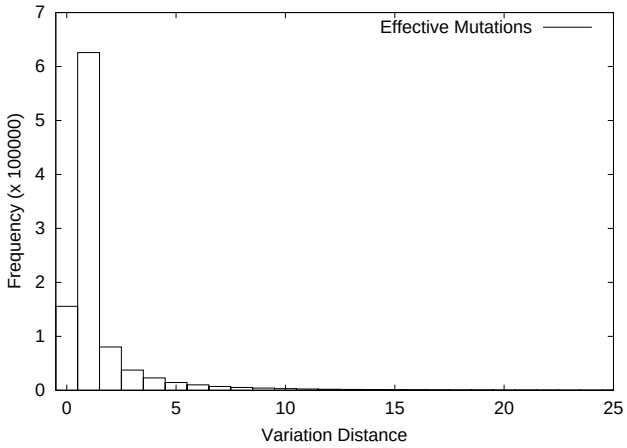


Fig. 1.7. Distribution of effective mutation distances for iris problem (similar to even-8-parity). Average figure over 100 runs.

Controlling variation step size on effective code level necessarily requires a distance metric that measures causal connections between program changes

and fitness changes sufficiently precisely. Even if already small modifications of the program structure may result in almost arbitrary changes in program behavior, smaller variations of the genotype should lead to smaller variations of the phenotype for a higher probability. In [13] this has been demonstrated by applying the edit distance metric to program trees. Figure 1.8 clearly shows a positive correlation between the mutation distance on code level and the mutation distances on fitness level for our code-selective metric on linear genomes. The corresponding distribution of mutation distances in Figure 1.7 confirms this to be true for the vast majority of occurring distances.

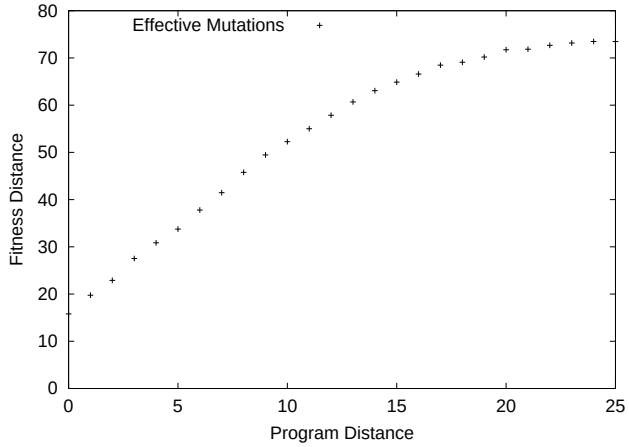


Fig. 1.8. Correlation between program distance and fitness distance for iris problem (similar to even-8-parity). Average figure over 100 runs.

The first observation, that small mutation steps occur more frequently than large mutation steps, fulfills guideline M 2 of a metric-based evolutionary algorithm (MBEA, see Section ??). The second observation (causality) forms a necessary precondition for the success of evolutionary algorithms in general.

1.7 Control of Structural Diversity

The effective distance metric between programs from Section 1.6.1 is applied here for an explicit control of genotype diversity, which is the average structural distance between individuals in the population [7]. Therefore, we introduce the two-level tournament selection shown in Figure 1.9. On the first level, individuals are selected by fitness (fitness selection). On the second level, the two individuals with maximum distance are chosen among three fitter individuals (diversity selection). More precisely, we select for the

effective edit distance minus the distance of effective lengths.[4] By doing so, the growth of effective length is not rewarded directly during selection.

While an absolute measure, such as fitness may be compared between two individuals, selection by a relative measure, such as distance or diversity necessarily requires a minimum of three individuals. In general, two from n individuals are selected with the greatest sum of distances to the $n-1$ other individuals. Selection pressure on the first level depends on the size of fitness tournaments. Pressure of diversity selection on the second level is controlled by the number of these tournaments. Additionally, we use a selection rate parameter in order to tune the selection pressure on the second level more precisely.

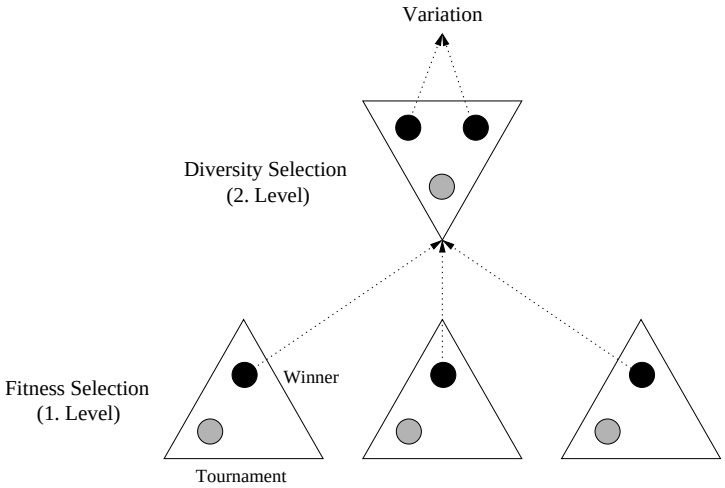


Fig. 1.9. Two-level tournament selection.

The two-level tournament selection constitutes a multiobjective selection method that finds individuals that are fitter and more diverse in relation to others while selection for fitness keeps the higher priority. Selecting individuals only by diversity for a certain probability instead does not necessarily result in more different search directions among better solutions in the population. Dittrich et al. [11] report on a spontaneous formation of groups when selecting the most distant of three individuals from a population of real numbers.

The number of fitness calculations and the processing time, respectively, do not increase with the number of tournaments if the fitness of individuals is saved and is updated only after variation. Only diversity selection itself becomes more computationally expensive the more individuals participate in

[4] This is possible because both measures, edit distance and length distance, operate on instruction level here.

it. n individuals require $\binom{n}{2}$ distance calculations. Hence, program distance should be efficiently computable.

A fitness landscape on the search space of programs is defined by a structural distance metric between programs and a fitness function that reflects the quality of program semantics. The application of a genetic operator corresponds to performing one step on this landscape. The active control of structural diversity increases the average distance of individuals. Graphically, the individuals spread more widely 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.

Note that the two-level selection process can also be used for an explicit control of code growth (complexity selection). Therefore, in linear GP a selection pressure may be imposed specifically on the smallest effective or non-effective program length.

1.7.1 Prediction Performance

For the two test problems introduced in Section 1.6, Table 1.5 summarizes average error rates obtained with and without selecting for structural diversity. Different selection pressures have been tested. For the minimum number of fitness tournaments (three) we used selection probabilities 50 percent and 100 percent. Higher selection pressures are induced by increasing the number of tournaments (up to four here). For each problem and variation operator, the performance increases continuously with the influence of diversity selection. The highest selection pressure that has been tested results in about a 2 to 3 times better prediction error on average, than without increasing diversity actively.

Again, as already demonstrated in Section 1.5 for regression tasks, linear GP performs significantly better by using effective macro-mutations instead of crossover. Obviously, the linear program representation is more suitable for being developed by mutations only, especially if those concentrate on effective instructions. Nonetheless, the experiments with linear crossover show that diversity selection does not depend on a special type of variation. Moreover, the application of this technique is demonstrated with a population-dependent operator.

1.7.2 Development of Diversity

Figures 1.10–1.13 illustrate the development of structural diversity during runs for different selection pressures and different variation operators. Obviously, the higher the selection pressure is adjusted, the higher is the diversity. Even without applying diversity selection, the average effective program distance does not drop towards the end of runs. While the diversity of effective code increases with crossover until a certain level and stays rather constant then, diversity with effective mutations increases more linearly.

Table 1.5. Second-level selection for structural diversity with different probabilities and different numbers of fitness tournaments (#T). Average error over 200 runs. Statistical standard error in parenthesis. Percentage difference from baseline results.

Variation	Selection		iris		even-8-parity	
	%	#T	*mean* (*std*)	Δ (%)	*mean* (*std*)	Δ (%)
cross	0	2	2.11 (0.10)	0	58 (3.4)	0
	50	3	1.42 (0.08)	33	35 (2.4)	40
	100	3	1.17 (0.07)	44	27 (2.2)	53
	100	4	1.09 (0.07)	48	19 (1.8)	67
effmut	0	2	0.84 (0.06)	0	15 (1.2)	0
	50	3	0.63 (0.05)	25	12 (1.0)	20
	100	3	0.60 (0.05)	29	10 (1.1)	33
	100	4	0.33 (0.04)	61	7 (0.8)	53

Two major reasons can be found to explain this behavior: First, genetic programming is working with a variable length representation that grows continuously during a run. In linear GP this is especially true for the effective program length, which may still grow even if the absolute length has reached the maximum. The longer effective programs become the bigger effective distances are possible. Actually, the growth of effective code is restricted earlier with crossover because of a much higher proportion of noneffective

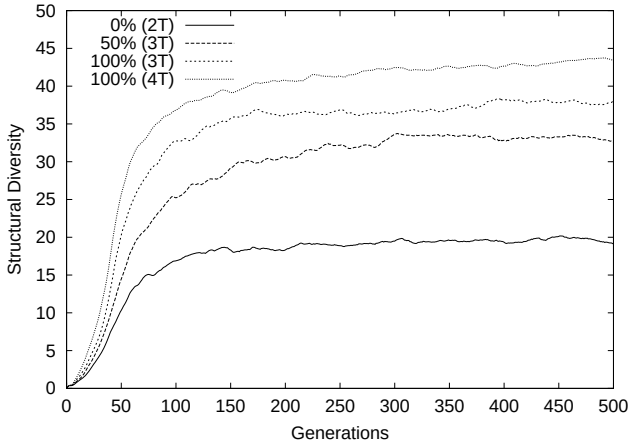


Fig. 1.10. Iris problem: Diversity with crossover and different selection pressures. Selection pressure controlled by selection probability and number of fitness tournaments (T). Average figures over 100 runs.

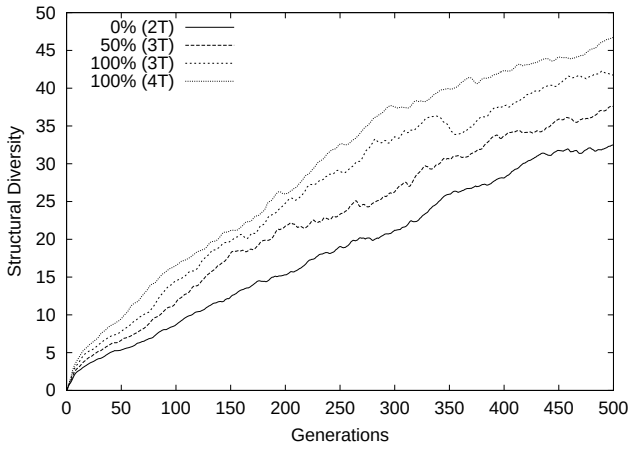


Fig. 1.11. Iris problem: Diversity with effective mutations and different selection pressures. Selection pressure controlled by selection probability and number of fitness tournaments (T). Average figures over 100 runs.

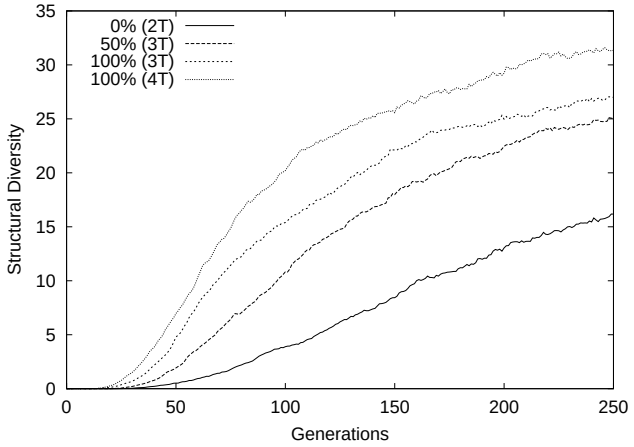


Fig. 1.12. Even-8-parity problem: Diversity with crossover and different selection pressures. Selection pressure controlled by selection probability and number of fitness tournaments (T). Average figures over 100 runs.

code that emerges with this operator (approximately 50 to 60 percent here). It is important to note that the average effective program length is hardly influenced by the distance selection in comparison to the average program distance.

Second, both forms of variation, linear crossover and effective mutation, maintain program diversity over a run implicitly in terms of the structural distance metric. This is true even without the explicit selection for diversity.

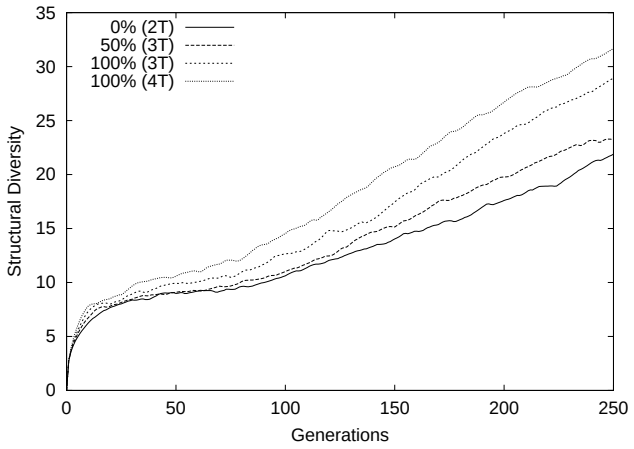


Fig. 1.13. Even-8-parity problem: Diversity with effective mutations and different selection pressures. Selection pressure controlled by selection probability and number of fitness tournaments (T). Average figures over 100 runs.

For linear crossover the reason might be in its high variation strength. Additionally, the high amount of noneffective code contributes to a preservation of effective code diversity with this operator. When using mutations exclusively instead, a high degree of innovation is introduced continuously into the population. This leads to a higher diversity of effective code than it occurs with crossover (see Figures 1.10–1.13) in consideration of the fact that the average effective length is about the same here for crossover and effective mutations in the final generation.

The stronger one selects for diversity, however, the more diversity gains ground in crossover runs. Obviously, a stronger influence on population diversity can be observed with crossover than with effective mutations. In comparison to mutation the success of recombination depends more strongly on the composition of the genetic material in the population. The more different two recombined solutions are, the higher is the expected innovativity of their offspring.

In the previous sections we have concentrated on improving the linear GP approach. In particular, the development of methods has been based on the distinction between used and unused parts of the linear representation. In tree programs, by comparison, redundant substructures may emerge only from program semantics, not from program structure.

In the following sections we will apply genetic programming for solving different problems from machining technology. Even if, in principle, linear GP may be used as well as tree-based GP in this problem domain, we have decided on the latter approach here. An introduction to this standard GP

variant is given along with a description of the problem specific configurations that have been chosen here.

1.8 Genetic Programming in Machining Technology

Modeling the chip-building process in cutting has been in the center of interest for a long time. The chip geometry, the material movement and the thermal processes that take place at the center of the cutting zone are decisive for high quality machining, reduction of machining times, and tool wear. Most existing approaches incorporate finite element or molecule dynamic methods as well as analytical techniques based on cutting force models [1,14,23,25,29,37] (see Figure 1.14). CI methods are used for this problem as

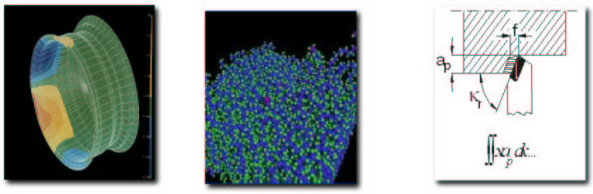


Fig. 1.14. Finite element (left), molecular dynamics (center) and analytical (right) methods.

well [20]. These approaches focus on neural networks. Neuronal networks can model the general behavior of correlations, but the models cannot be used aside from neuronal networks for further usage or investigations.

Another CI method is symbolic regression via genetic programming. Symbolic regression is a widely used method for reconstructing mathematical correlations. Here a new graphical representation of the individuals is also presented. This new three-dimensional representation allows the user to recognize certain possibilities to improve the setup of the process parameters. Furthermore, this new representation allows the visualization of the generated three-dimensional objects with nearly every CAD program for further use.

The approach of symbolic regression using genetic programming is quite similar to that of a human observer [29] watching a certain process and trying to describe it in physical terms. Furthermore, the system may produce some unexpected but valid results, which do not correspond to any known properties of the process. If such phenomena occur, new knowledge about the process has been generated. This knowledge may be helpful in supporting the development of new simulation tools, which are able to increase the productivity

of the process as well as its reliability. In chip modeling, these models can be described in a two dimensional way by relations of this type:

$$\boldsymbol{f}(t) = \begin{pmatrix} f_x(t) \\ f_y(t) \end{pmatrix} \tag{1.5}$$

To succeed in this objective, some problems have to be solved. The first task is to obtain the trajectories of the particles in the crystalline structure of the metal workpiece, shown in Sections 1.9 and 1.10. The second is to find a suitable model describing the physical correlations that lead to these trajectories. For this task the method of symbolic regression via genetic programming is used [2,16]. This is shown in Section 1.11.

1.9 Experimental Setup

The primary goal of the experiments was to gather some exemplary data from the processes. In order to show the power of the developed algorithms, several different processes and methods of image processing have been tried. They all have a similar principle setup in common. The first step consists of taking pictures of the process without disturbing the cutting process or influencing the chip formation. This step is done on two different experimental setups to gather a wide range of data. The second step is to transform these pictures into a digital form.

1.9.1 Extracting Particle Trajectories from a Turning Process

The first experimental setup was used on the turning of aluminum alloys. This setup was similar to that described by Warnecke [29]. In contrast to his setup, no microscope is used. In this experiment a high speed camera, Weinberger Speedcam+ , which takes up to 4500 pictures per second, was used. A schematic view of this setup is shown in Figure 1.15. A Grob BZS-600 turning machine was used for this process to cut the aluminum rings. The whole chip-building process was filmed with the high-speed camera. About 2 seconds of every run were recorded. The experiment was carried out with various process parameters. Two different alloys were used as workpiece material. The workpiece was a ring-shaped profile, which was rotating while the tool was fixed. The cutting speed $v_c[\frac{m}{min}]$, the feed $f[mm]$, and the back engagement of the cutting edge in $a_p[mm]$ were varied.

1.9.2 Extracting Particle Trajectories from a Drilling Process

The second setup to receive data from a cutting process was a modified drilling process. A short, rigid, and torsion-resistant drill was used. Therefore the workpieces could be formed as cylinders with a slightly reduced diameter compared to the diameter of the drill. Thus, the high-speed camera could be

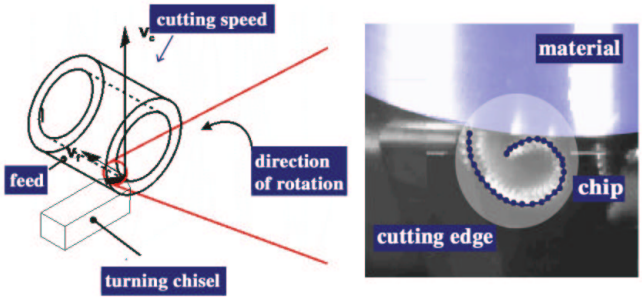


Fig. 1.15. Experimental Setup for turning of aluminum alloys.

Table 1.6. Parameters used for the turning process.

Indexable Insert (Seco)	workpiece	$v_c[\frac{m}{min}]$	$f[mm]$	$a_p[mm]$
DNMG 150608-MF2 TP200	9SMn28k	265	0,20	4
DNMG 150608-MF2 TP200	9SMn28k	265	0,50	4
DNMG 150608-MF2 TP200	9SMn28k	380	0,20	4
DNMG 150608-MF2 TP200	9SMn28k	380	0,40	4
DNMG 150608L-95 HX	AlCuMgPb	500	0,05	2
DNMG 150608L-95 HX	AlCuMgPb	500	0,45	2
DNMG 150608L-95 HX	AlCuMgPb	50	0,05	4
DNMG 150608L-95 HX	AlCuMgPb	50	0,40	4
DNMG 150608-MF2 TP200	C60	50	0,08	2

positioned at the side of this cylinder to enable the filming of the chip flow at the cutting edge of the workpiece. Figure 1.16 shows a schematic view of this setup as well as a photo of the machine (NBH70 from Hüller-Hille). Due to its rigid design, this machine allows very low turning speeds without loosing precision. In this setup only the feed was varied. One difficulty was to determine optimal parameters for the camera setup to obtain the best possible images of the process. For this purpose the parameters of the camera were also varied (Table 7.6). The parameters are the frequency of the image recording and the intensity of the illumination by the stroboscopic lights. The pictures that were taken in this drilling experiment were edited afterwards, using image processing software to increase contrast and thereby the visibility of the chip flow. Two different types of chips occur in this setup because of the two cutting edges of the used drill. Figure 1.17 shows 6 frames extracted from the processed movie. All pictures show the work of the inner cutting edge. Low cutting speed at the inner cutting edge leads to a shear chip formation. It can be seen how the chip is sheared and pushed together. The effect of the chipbreaker on the chip can be seen.

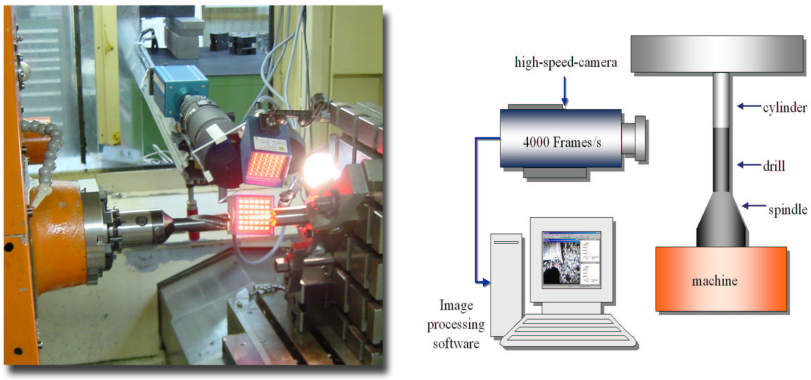


Fig. 1.16. The used machine (left) and a schematic view of the setup (right).

Table 1.7. Parameters used for the drilling process.

$v_c[\frac{m}{min}]$	$f[mm]$	$f[\frac{1}{s}]$
50	0,40	1000
50	0,40	2000
50	0,15	4000
100	0,15	1000
150	0,15	1000
150	0,15	2000
150	0,15	4500

1.10 Gathering Data

After gathering the image data from the different cutting processes significant trajectories of particles in the material need to be extracted. This was done in two steps. First, a segmentation of the particles in the filmed material was carried out. Afterwards, the trajectories of the particle flow were tracked. This can be done either manually or automatically, depending on the type of the material. In filmed sequences, an automatic tracking method is useful to reduce the manual effort.

1.10.1 Segmentation

The task of determining which pixels of an image belong to a single element of the filmed object is called segmentation. Here, single particles of a metallic workpiece have to be located to simplify the tracing of the flow in the cutting process. The problem is to find sets of points in the image that belong to a single crystalline or another solid part in the flow. This can be any part

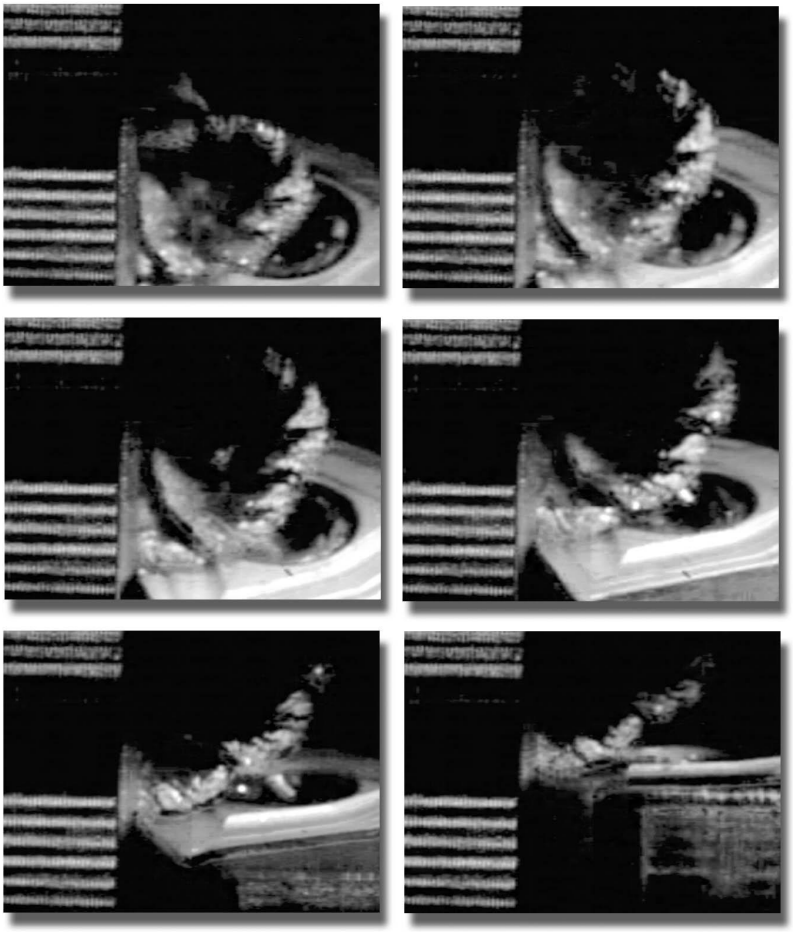


Fig. 1.17. Shear chip formation at the inner edge of the drill filmed at $4500 f/sec$.

of the chip. All particles in the flow which follow significant paths are suitable. Therefore, the main difficulty is to find the same arbitrarily chosen but significant particle in every frame of the filmed data.

1.10.2 Automatic Extraction

Automatic extraction of trajectories from filmed material requires a certain quality of the film sequences to enable a software to trace the particles. Therefore, the software used for this tracking needs to be able to recognize the same particle in different frames of the filmed material. Traced particles need to

be on the filmed surface during the whole process of tracking. Otherwise the particles need to be segmented on every frame of the filmed material.

In oder to extract the particle trajectories, the program WINAnalyze from Mikromak was used. Provided that the images had a certain quality, the software was able to follow a segmented particle automatically. A screenshot of the software performing a full automatic segmentation is shown in Figure 1.18.

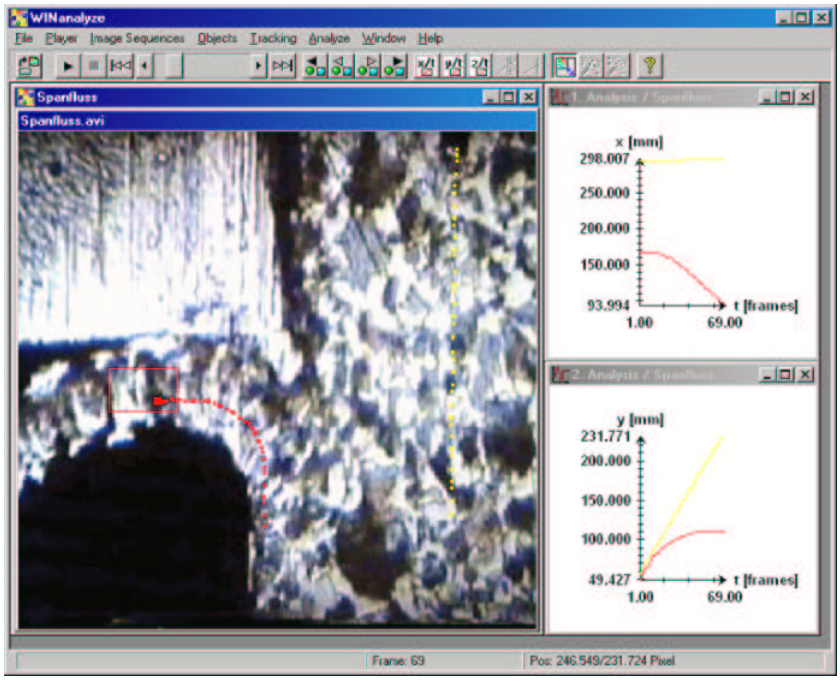


Fig. 1.18. The software WINAnalyze v.13 (Mikromak).

The program generates ASCII trace files, which can analyze the trajectories of the particles by the GP system. The positions of the particles can now be interpreted as the values of a two-dimensional relation.

In addition to the fully automatic method, manual segmentation was used. This method was used when, due to optical or material deformation effects, it was not possible to locate one single particle in every frame automatically. After this step, the software can carry out the automatic extraction of the trajectories. This semi automatic method was used on the data extracted from the turning process as described in Section 1.9.1.

1.10.3 Manual Extraction

As a second method, a manual tracing method was used to extract the particle trajectories. This was used in cases where the extraction of the particle dynamics could not be obtained automatically due to optical or material deformation effects. These data from the drilling process, (see Section 1.9.2) had to be prepared with this fully manual method, due to the strong deformation of the chip volumes and the low image resolution resulting from the high frame rate of the movies (up to 4500 images per second), see Figure 1.19.

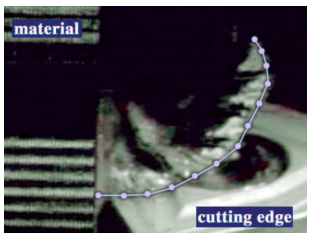


Fig. 1.19. The manually analyzed chip volumes generated in the drilling process.

1.11 Tree Based GP for Symbolic Regression

Symbolic regression via genetic programming was used to model the physical correlations that underlie the processes shown in Section 1.9. The implementation of the genetic programming kernel contains several aspects from evolution strategies [24] and genetic programming [16] as well.

1.11.1 GP System

The evolution starts with a randomly generated set of functions (initial population). Due to the fact that the algorithm has to evolve parametric functions, two symbolic representations have to be generated in parallel. The genetic operations are applied to this first generation of functions and a succeeding generation is produced, (see Figure 1.20). The fitness of an individual is measured directly by evaluating the evolved formulae and comparing the geometric shape of these parametric functions with the given point set i.e., the trajectory of the chip. Technically the formulae are represented as instances of a tree-based data structure, which is implemented in C++ using the Standard Template Library [28]. Hence, the fitness function can be evaluated in a fast and efficient way. It is also possible to use an interpreter, which may be more flexible in some cases (e.g., for debugging tasks), but

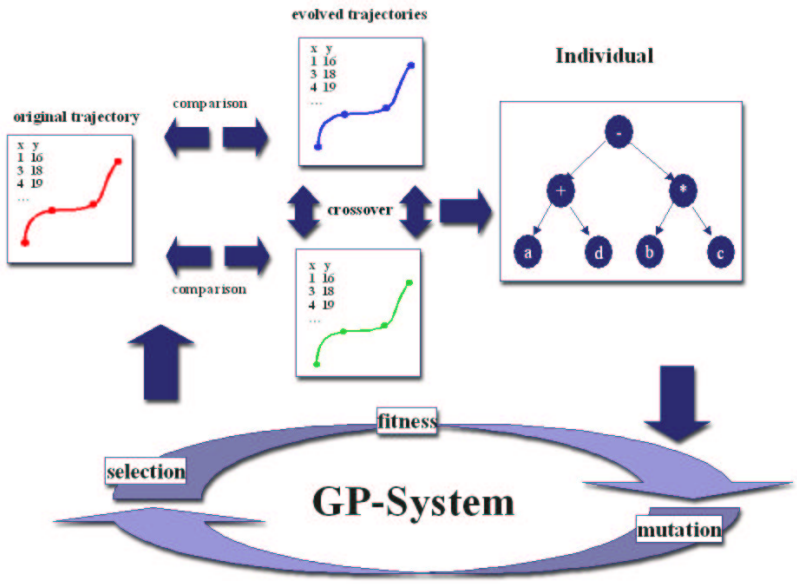
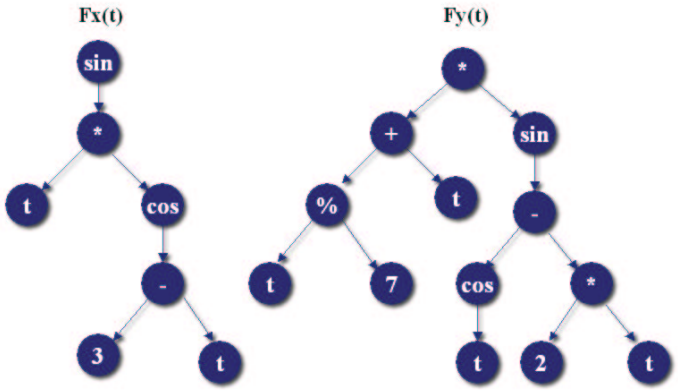


Fig. 1.20. General GP scheme.

slows down the evaluation. The structure of the program follows a basic GP scheme [16,15,19]. The probability of choosing a constant during mutation, the maximum size of an individual or the probability of changing a function or a terminal and the number of changes is defined in a parameter file. The data structure and the genetic operators will be described next.

1.11.2 Data Structure

In analogy to the infix notation of mathematical functions a tree based representation forms the genotype. A sample individual, consisting of two functions $f_x(t)$ and $f_y(t)$, is shown in Figure 1.21. The primary goal of obtaining the mathematical representations of two-dimensional curves leads to the phenotypic representation of an individual as graphs of a parametric function. This difference between genotypic and phenotypic representation and the need for genetic operators in which small changes in the genotype results in small changes in the phenotype yields two problems. First a set of genetic operators which allows to perform these small changes on the genotype must be used. Second a fitness function which correctly evaluates the fitness of one individual has to be defined.



$$f_{Indiv}(t) = \begin{matrix} f_x(t) \\ f_y(t) \end{matrix} = \begin{matrix} \sin(t * \cos(3-t)) \\ t/7 + t * \sin(\cos(t) - 2 * t) \end{matrix} \tag{1.6}$$

Fig. 1.21. Sample structure of a single individual and the corresponding functions.

1.11.3 Genetic Operators

The genetic operators are known as mutation and recombination. Mutation takes place on a single individual and changes one or more positions of the individual, here, a single node of the tree. In contrast to classical GP [2] inner nodes - as well as terminal nodes can be mutated in real, value steps. In Figure 1.22 mutation is illustrated by a single mutational tree insertion of a terminal node with value 8. To allow insertion of new nodes without erasing older nodes the randomly (out of all possible operators) chosen operator x+z was inserted together with the new node. The second genetic operator is the recombination between two arbitrary individuals (see Figure 1.23). The genetic recombination operator selects an arbitrary node in the first individual and replaces it by an arbitrarily selected node from the second individual. These two genetic operators allow a change in the genotype, yielding small changes in the phenotype of the individual. The amount of change in the phenotype can be reduced and increased by setting the amount of mutation (e.g., number of nodes to be chosen).

A closer look at these operators is given in [31,32,34,36].

1.11.4 Deterministic Correction of the Individuals

In order to improve the efficiency of the naive approach, some variations have been implemented [31]. This leads to a faster adaption of the curves without adding any extra knowledge to the algorithm.

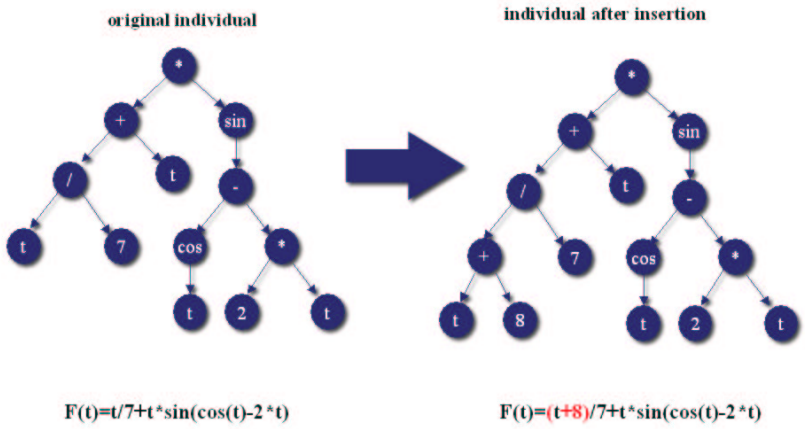


Fig. 1.22. Mutational insertion on a individual.

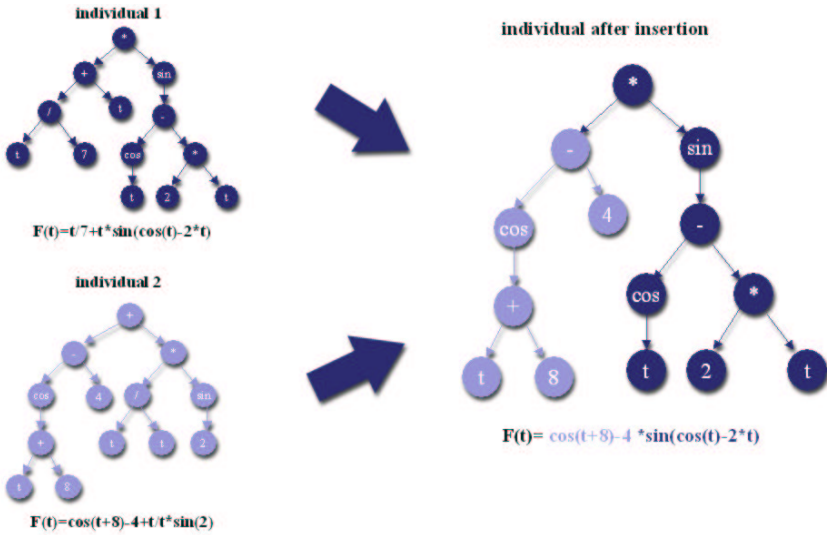


Fig. 1.23. Recombination of individual 1 with individual 2.

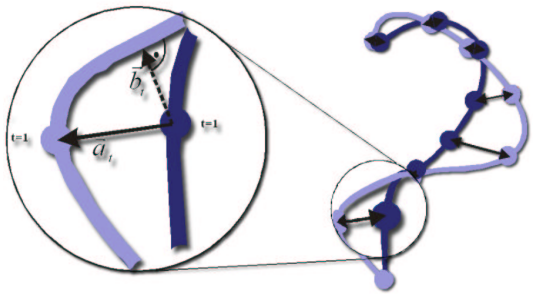
Due to the fact that many solutions of the GP algorithm show shapes that are similar to the correct solution but differ only in orientation and size, a deterministic adaptation step is inserted between the mutation and the fitness calculation of the individual.

First, a translation vector is determined by placing the starting point of the individual at the position of the first point of the trajectory. The

translation is performed by adding to each point of the genetically generated function. In a second step, the individual will be scaled to the size of the trajectory curve. The scaling values are determined by setting the maxima of the individual to the same value as the maxima of the trajectory points. This leads to a faster adaption of the curves without adding any extra knowledge to the algorithm.

1.11.5 Multi Objective Fitness Function

Here (Figure 7.24) a multi-objective fitness function that compares the function values of the individual with corresponding points of the trajectory is used. If all points of the trajectory lie on the function plot this individual will represent a perfect solution to the problem. The fitness values will be represented by a weighted point-to-value distance scheme (1.7), and they will be summed up by a root mean square sum.



$$F_{dist} = \sum_{t=0}^{t_{max}} (|\vec{a_t}|) \tag{1.7}$$

Fig. 1.24. The used point to value distance scheme.

1.11.6 Weighting the Size of the Individuals

Individuals in genetic programming can grow very rapidly [16,18]. This is referred as code bloat and can be observed in linear genetic programming or in tree-based genetic programming. Quadratic [17] and exponential growth are documented [22]. To stop individuals from growing and thus, from slowing down the evaluation, an extra weighting function is inserted into the fitness function. Both fitness values, the weighted point to value distance scheme (1.7) and the weighted number of nodes, are combined- 1.8

$$F_{indiv} = F_{dist} * \delta_w - F_{node} * \nu_w \tag{1.8}$$

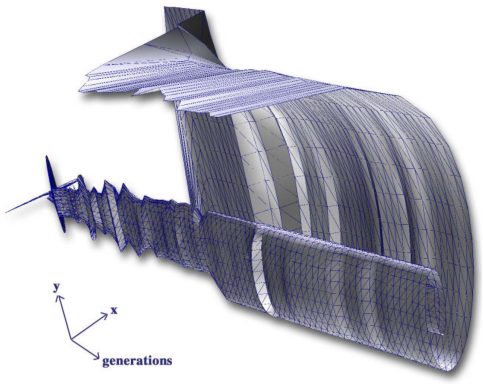


Fig. 1.25. Individual extruded in a space-time plot.

$$F_{indiv} = \sum_{t=0}^{t_{max}} (|\vec{\boldsymbol{a}_t}|) * \delta_w - (((\alpha) - (\alpha_e))^4) * \nu_w \tag{1.9}$$

Table 1.8. The parameters are here.

parameter	description
δ_w	distance weight $[0-1]$
ν_w	number of nodes weight $[0-1]$
α	number of nodes of an individual $[>0]$
α_e	estimated optimal number of nodes $[>0]$

The estimated number of nodes is added to the function to force the solution into the global optimum. To verify this procedure a set of tests have been carried out. These tests have been performed to determine the sufficient weighting of a punishment of a node count. The results of these tests are shown in Section 1.13.3.

1.12 Graphical Representation

Due to the fact that choosing the right parameters for setting up a fast and safe reconstruction of the estimated functions is difficult, a new graphical representation of a complete GP run was developed. In order to allow an overview of the whole chronology of the reconstruction process [33], a third dimension was added to the two dimensions (function values of $f_x(t)$ and

$f_y(t)$) of the current best individual of the run (see Figure 1.25). This third dimension is the index in the order of the arrival of the best individuals in the reconstruction. This is achieved by dumping all best individuals in this current run, ordered by the time of their appearance. Therefore, a standard CAD -CAM file format was used. This stereo lithography (STL) format is well defined and can be used in various CAD -CAM systems or computer-graphic systems. This opens various possibilities for rendering and later usage of the generated objects. The data is generated by building triangles that have two of their vertices in two consecutive points of one individual and one vertex in another neighboring individual. That is, one triangle consists of the points $p[in][pn]$ and $p[in][pn+1]$ in the individual indicated by in and a third point $p[in-1][pn]$ in a second individual indicated by $in-1$. It is obvious that, in order to get a closed surface, the following triangle has two points in the individual indicated by $in-1$, all three points for this individual are $p[in-1][pn]$, $p[in-1][pn+1]$ and $p[in][pn+1]$. This scheme is shown in Figure 1.26. In Figure 1.25 a sample rendering of such a plot is

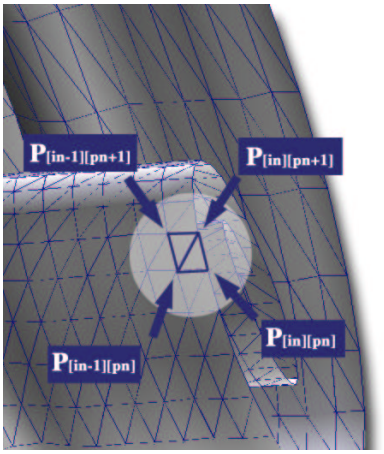


Fig. 1.26. The scheme for the triangulation at two triangles.

shown. The rendering was carried out using a third party software called 3D Exploration from Righthemisphere, Inc. The timeline goes from left (back) side of the figure to final stage at the right hand-side (front). In this final stage the individual has reached a fitness of 0.98. One thing that can be seen from this picture is that only a small number of steps is needed to reconstruct the general shape of the individual and that after this phase the algorithm behaves more like an evolution strategy [24] for reaching the final shape.

1.13 Results of the GP Kernel

1.13.1 Test Function

In this case a known function (1.10) was used in order to illustrate the operation of the symbolic regression algorithm

$$\boldsymbol{f}(t) = \begin{pmatrix} f_x(t) \\ f_y(t) \end{pmatrix} = \begin{pmatrix} \sin(\frac{\pi}{180} * t) * \frac{t}{10} \\ \cos(\frac{\pi}{180} * t) * \frac{t}{10} \end{pmatrix} \tag{1.10}$$

. The shape of the graph of $\boldsymbol{f}(t)$ resembles a chip. The values of the parameter t range from 0 to 360. For simulation purposes this function was sampled yielding discrete points, which represent points that could have been sampled from a film sequence. The final stage of the reconstruction, the plot and the graph of the best individual, is shown in Figure 1.27. In spite of weighting

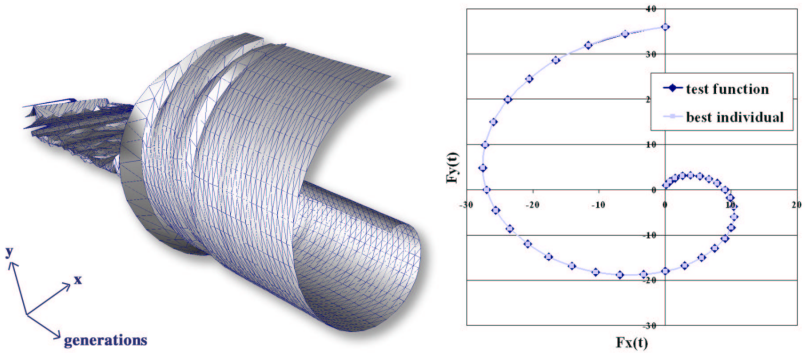


Fig. 1.27. The final stage of the reconstruction of the test function.

the length of the individual in the fitness calculation, the relation function (1.11) of the best individual after this reconstruction contains more nodes than the original function (1.10).

$$\boldsymbol{f}(t) = \begin{pmatrix} \frac{\sin(\frac{t}{57.2969}) * \frac{t}{27.8692}}{\sin(25.4997)} \\ \cos(\frac{t * \sin(149.902)}{44.6694}) * 12.2718 * \sin(40.5023) * \frac{t}{40.738} \end{pmatrix} \tag{1.11}$$

After the resulting function has been simplified by hand (1.12) it can be seen that the real structure of the searched relation function is found by the algorithm. This basic structure can be used for further extrapolations. The reduced form is:

$$\begin{matrix} f_x(t) \\ f_y(t) \end{matrix} = \begin{pmatrix} \sin(0.0175 * t) * t * 0.0920 \\ \cos(0.0178 * t) * t * 0.1000 \end{pmatrix} \tag{1.12}$$

Another way to analyze the behavior of the population is to store a snapshot of the whole population into one figure. All individuals are sorted in order of their fitness values to enable a closer look at the progression of the solution within the whole population. Figure 1.28 shows such a plot of the reconstruction of the test function at particular times of the reconstruction process. Table 1.9 shows the parameters for this test run. It can be seen that due to the application of a $(\mu + \lambda)$-selection strategy taken from evolution strategies [24] in the regression process, the best individual propagates very fast through the whole population. The use of a type of a parallel multi population strategy may lead to a higher genetic diversity (see Section 1.14).

Table 1.9. Parameter table for diversity test.

value	parameter
200	Number of parents per generation.
300	Number of children per generation.
0.001	Weight of the node count fitness to overall fitness.
1	Weight of the distance fitness to overall fitness.
10	Estimated best node count in this reconstruction.

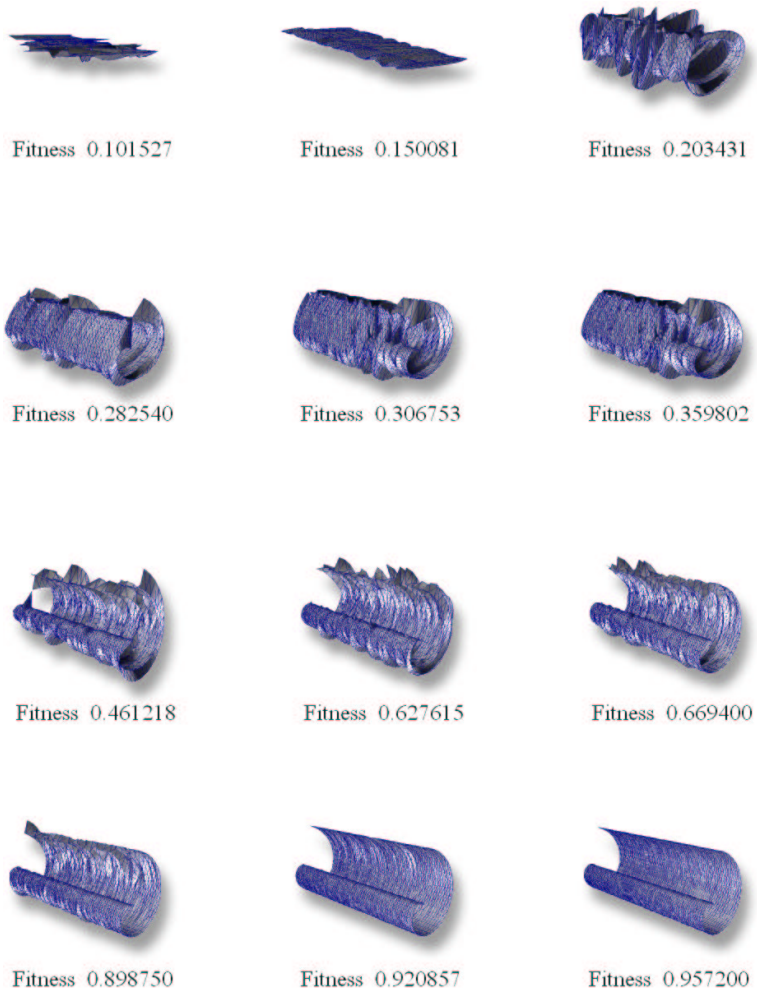


Fig. 1.28. The population at different generations of the reconstruction. Each picture shows one complete population at discrete values of their overall fitness. The individuals are sorted according to their fitness from lower fitness at the left to greater fitness at the right side of the 3D shapes. During the starting phase where low fitness values appear a strong diversity of the individuals in the population can be observed. With increasing fitness values the population looses its diversity. At this stage the main reconstruction work is performed by variation of the similar individuals.

1.13.2 Real World Data

After the verification of the GP scheme using a known test function, the algorithm is applied to data extracted from a filmed sequence of the turning process, as described in Section 1.15. To get information about the physical correlations that lead to the shape of the chip in the turning process, the positions of the chip are manually segmented (see Figure 1.29). One difference

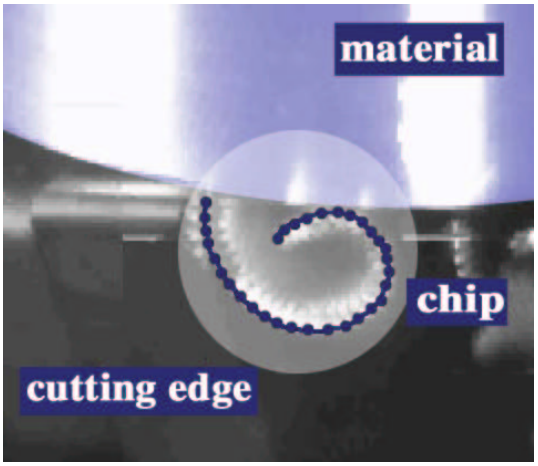


Fig. 1.29. Manually traced particle flow (picture size 30 mm by 30 mm).

in the preceding test function is the lack of knowledge of the exact number of nodes needed to obtain a good model of the physical process. Therefore, the number of necessary nodes has to be estimated, and as a consequence, the weighting factor of the number of nodes in the fitness calculation had to be reduced. The parameters for the estimated best node count are increased to 20 nodes, and the average initial size of the individuals is increased to 20 nodes. Therefore, the resulting relation functions (1.13) are not as short and handy as the results from the test function. Figure 1.30 shows that the algorithm has a tendency to smooth the trajectories. The approximations can be improved locally by increasing the number of sampling points. This way the reconstructions are forced to follow also small bendings of the particle flow.

$$\boldsymbol{f}(t) = \begin{pmatrix} \cos\left(\frac{-0.8995}{\sin\left(\frac{\cos(102.448)}{\frac{t+40.5886}{-23.1577}}\right)}\right) * t + 3 * t * \sin\left(\frac{\frac{0.447}{t}}{0.0163}\right) + 0.3679 * t \\ 0.709 + \frac{t+\sin(t)+\frac{0.020}{t}-1655.335*t}{\sin(t)+148.944-t} - t * 0.624 + 0.432 \end{pmatrix} \quad (1.13)$$

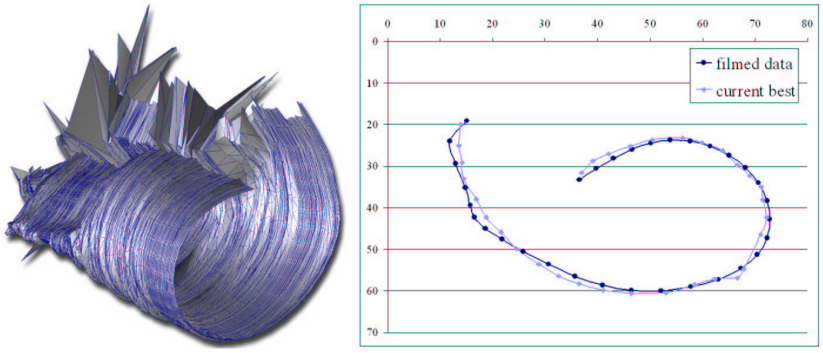


Fig. 1.30. Three dimensional plot of best individual and the original data.

1.13.3 The Dependency of the Number of Nodes

To evaluate and to verify the strategy of weighting the lengths of the individuals (see Section 1.11.6), several tests were performed. For the tests, two different target functions to the parameters were used. About 400 single reconstruction runs were carried out per series.

Table 1.10. Parameter table for both test series.

value	parameter
30	Number of parents per generation.
60	Number of children per generation.
1	Weight of the distance fitness to overall fitness.
20	Estimated best nodecount in this reconstruction.
5000	Maximum number of generations.
20	Average size of new individuals.
0.9	Probability of inserting a node.
0.9	Probability of deleting a node.
0.9	Probability of mutation a inner node.
0.6	Probability of inserting a value.
0.5	Probability of inserting a function.
10	Amount of mutation to a value.
-100	Maximum value of a new value node.
100	Minimum value of a new value node.

Two parameters were varied to examine their influence on the fitness evaluation. The value for the estimated best number of nodes per individual was varied from 0 to 20. To examine the influence of the weighting of the number of nodes, the weighting factor in every run was estimated by the result of $1/d$ where d was increased from 0 to 20 over the whole test run. In Figure 1.31 the first two plots at the top are taken from one test series

that was done using the known test function. The two plots at the bottom are taken from a second run with the real-world relation extracted in the experiment described in Section 1.9.1.

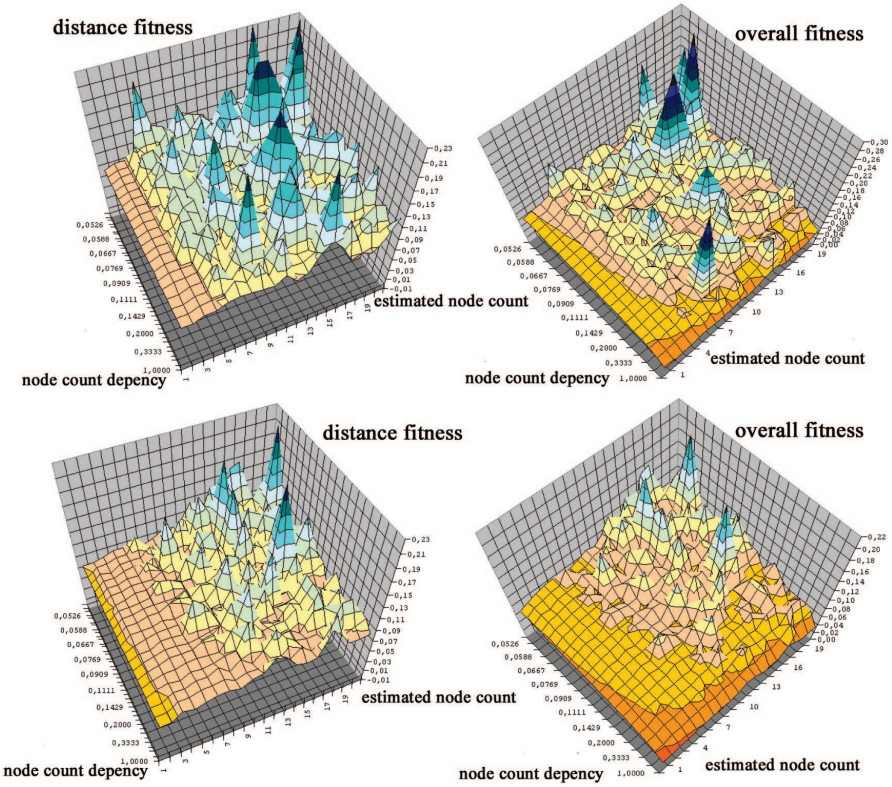


Fig. 1.31. Fitness of the test function (top) and the real-word data (bottom), distance fitness F_{dist} (left), overall fitness F_{indiv} (right).

1.14 Parallelization

To overcome the problem of low diversity in a population, a distributed parallel approach was implemented into the symbolic regression approach. The task here was to prove the possibility of speed improvement that could be achieved by multiple computer systems or processors [27].

The general idea of the parallelization is based on the experiences of [30] and [35]. For this reason a strong similarity between the algorithms used

in both applications see [31,34] was very useful for the implementation. As was done in [35], a blackboard communication model has been applied. Each instance of the population posts a certain number n of individuals to the blackboard every g generations. The parameters n and g are specific to each population. So the migration rate can be chosen according to the speed of the computer that carries the population. The implementation is based on file sharing mechanisms. This allows a decentralized and flexible parallelization.

1.14.1 Tests

Due to the splitting of the population, the problem of low diversity was reduced. First tests show noticeably higher convergence rates. As it can be

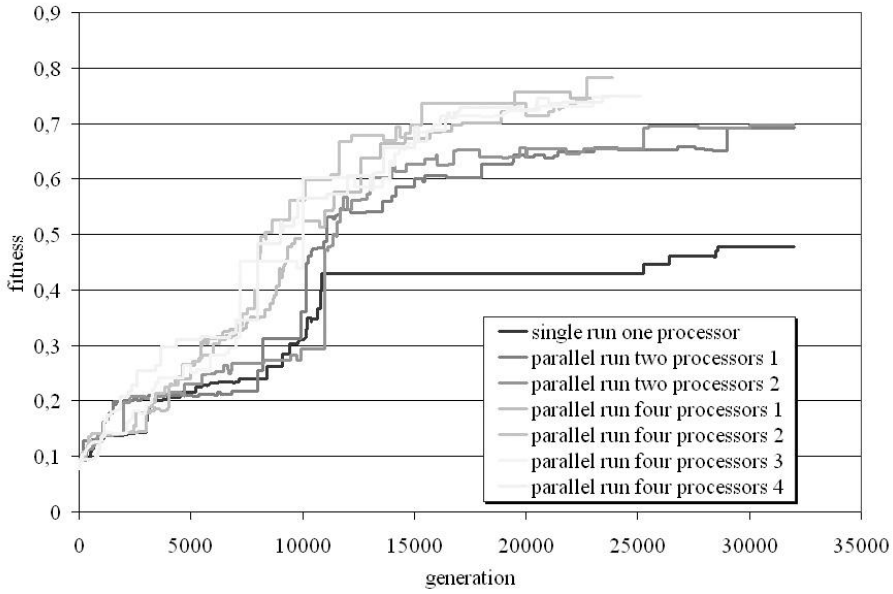


Fig. 1.32. Convergence tests on the distributed implementation.

seen in Figure 1.32, convergence increases, and the overall computing time decreases. A similar behavior can be seen in [35]. The parameters for all runs are equal, as shown in Table 1.11. Migration frequency, i.e., the number of generations after which a certain number of individuals migrate, should not be higher than 1/1000 to stop good individuals from spreading around all populations. This is similar to conclusions Cantu-Paz has published in [9]. It

Table 1.11. Parameter table for all parallel test series.

value	usage of this parameter
200	Number of parents per generation.
200	Number of childrens per generation.
0.000001	Weight of the nodecount fitness to overall fitness.
1	Weight of the distance fitness to overall fitness.
10	Estimated best nodecount in this reconstruction.
100000	Maximum number of generations.
10	Average size of new individuals.
0.9	Probability of inserting a node.
0.9	Probability of deleting a node.
0.9	Probability of mutation a inner node.
0.6	Probability of inserting a value.
0.5	Probability of inserting a function.
5	Amount of mutation to a value.

is presumed that this behavior again may result from the usage of the $(\mu + \lambda)$ strategy (see Figure 1.28).

Further investigations will be carried out to examine the behavior of different parameterizations in every different population. So the influence of different node weighting parameters may be interesting. Additionally, some examinations will be made regarding the behavior of a comma strategy to stop the decreasing diversity.

1.15 Conclusion

This chapter reported on genetic programming including advanced methodic aspects and applications to machining-technology problems. Development of methods concentrated on the linear program representation. In particular, the identification of effective and unused parts of code was addressed. On the basis of this information we could accelerate program evolution, not only in computing time but also on the level of generations. Efficient mutation operators were developed for linear programs that increased speed of fitness convergence significantly. Even though the mutation-based LGP variant provides a sufficient preservation of code diversity already implicitly, increasing the level of diversity explicitly by a multiobjective selection mechanism further improved prediction performance. In general, mutation-based variation was found to be much more successful than crossover-based variation in linear GP. The main reason is that mutations induce smaller step sizes on the linear program structure that allow a more continuous approximation to solutions of higher quality. Additionally, a code-selective distance metric was applied in order to reduce mutation step sizes more precisely on the effective code.

It has been shown that the use of symbolic regression is suitable for constructing and developing descriptive models for physical relations. The process of reconstruction can be automatized. Experiments have proved that the reconstruction of correlations that take place in the chip building process in metal cutting is possible. There are also results showing that the quality of the found relations allows the reconstruction of other, not only physical correlations. The correlations may be taken from other scientific fields where process models still do not exist or should be renewed. This may be in applied social studies or applied economics. Another possible field is the reconstruction of signals, such as audio signals, or the task of filtering disturbances from given signals of low quality. The quality of the models found is good enough to use them for extrapolation or for further analysis.

A new method for analyzing the behavior of reconstruction processes of mathematical functions by symbolic regression via genetic programming has been developed. This graphical representation of the reconstruction process can be used in multiple ways to optimize or visualize the whole reconstruction process. Furthermore, it can be used to investigate specific points in time in the reconstruction. Additional considerations can be made to integrate this system into standard symbolic regression libraries.

The symbolic regression application was used for a distributed parallel approach to speed up the regression and to increase the convergence probability. First investigations show the efficiency of a blackboard parallelization method. This allows a high grade of parallelization without getting too much overhead from communication. The application is portable and flexible, and can be used on various types of systems. Further examinations need to be done on the optimal parameterization of the migration rate and the parameters of the population.

In order to develop a physical model of the turning process, additional considerations have to be made. For example, adding parameters, such as the cutting speed or the tool angle to the reconstruction process. Furthermore, restrictions have to be developed to constrain the length of the reconstructed individuals.

References

1. D. Bähre, M. Müller, and G. Warnecke. Basic characteristics on cutting effects in correlation to dynamic effects. In *Technical Papers of the 25th North American Manufacturing Res. Conference*, pages 21–26, Lincoln, VT, 20–23 May 1997.
2. W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction. On the Automatic Evolution of Computer Programs and its Application*. dpunkt/Morgan Kaufmann, Heidelberg/San Francisco, CA, 1998.

3. C. L. Blake and C. J. Merz. UCI Repository of Machine Learning Databases [http://www.ics.uci.edu/~mlearn/MLRepository.html].
4. M. Brameier and W. Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26, 2001.
5. M. Brameier and W. Banzhaf. Effective linear program induction. Technical Report CI-108/01, Collaborative Research Center 531, University of Dortmund, 2001.
6. M. Brameier and W. Banzhaf. Evolving teams of predictors with linear genetic programming. *Genetic Programming and Evolvable Machines*, 2(4):381–407, 2001.
7. M. Brameier and W. Banzhaf. Explicit control of diversity and effective variation distance in linear genetic programming. Technical Report CI-123/01, Collaborative Research Center 531, University of Dortmund, 2001.
8. M. Brameier and W. Banzhaf. Explicit control of diversity and effective variation distance in linear genetic programming. In *Proceedings of the Fifth European Conference on Genetic Programming (EuroGP-2002)*, Kinsale, Ireland, 3–5 April 2002. (Accepted).
9. E. Cantu-Paz. Migration policies and takeover times in genetic algorithms. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99)*, volume 1, page 775. Morgan Kaufmann, San Francisco, CA, 13–17 July 1999.
10. K. Chellapilla. Evolving computer programs without subtree crossover. *IEEE Transactions on Evolutionary Computation*, 1(3):209–216, 1997.
11. P. Dittrich, F. Liljeros, A. Soulier, and W. Banzhaf. Spontaneous group formation in the seceder model. *Physical Review Letters*, 84:3205–3208, 2000.
12. D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, New York, 1997.
13. C. Igel and K. Chellapilla. Investigating the influence of depth and degree of genotypic change on fitness in genetic programming. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99)*, volume 1, pages 1061–1068. Morgan Kaufmann, San Francisco, CA, 13–17 July 1999.
14. T. Inamura. Brittle/ductile phenomena observed in computer simulations of machining defect-free monocrystalline silicon. *Annals of the CIRP*, 46:31–34, 1997.
15. R. E. Keller, J. Mehnen, W. Banzhaf, and K. Weinert. *Surface Reconstruction from 3D Point Data with a Genetic Programming/Evolution Strategy Hybrid*, chapter 2, pages 41–65. Advances in Genetic Programming 3. MIT Press, Cambridge, MA, 1999.
16. J. R. Koza. *Genetic Programming*. MIT Press, Cambridge, MA, 1992.
17. W. B. Langdon. Quadratic bloat in genetic programming. In D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 451–458. Morgan Kaufmann, San Francisco, CA, 10–12 July 2000.

18. W. B. Langdon, T. Soule, R. Poli, and James A. Foster. The evolution of size and shape. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, Cambridge, MA, 1999.
19. J. Mehnen. Evolutionäre Flächenrekonstruktion. PhD thesis, University of Dortmund, 2000.
20. M. Müller. Prozeßidentifikation beim Drehen mit Hilfe künstlicher neuronaler Netze. *FBK - Produktionstechnische Berichte*, 22, 1996.
21. P. Nordin. A compiling genetic programming system that directly manipulates the machine-code. In K.E. Kinnear, editor, *Advances in Genetic Programming*, pages 311–331, MIT Press, Cambridge, MA, 1994.
22. P. Nordin and W. Banzhaf. Complexity compression and evolution. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 310–317, Morgan Kaufmann, San Francisco, CA, 1995.
23. H. Schulz and K. Bimschas. Optimisation of precision machining by simulation of the cutting process. *Annals of the CIRP*, 1993.
24. H.-P. Schwefel. *Evolution and Optimum Seeking*. Wiley, New York, 1995.
25. S. Shimada, N. Ikawa, H. Tanaka, and J. Uchikoshi. Structure of micromachined surface simulated by molecular dynamics analysis. *Annals of the CIRP*, 1994.
26. T. Soule, J. A. Foster, and J. Dickinson. Code growth in genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Proceedings of the Genetic Programming Conference (GP'96)*, pages 215–223, MIT Press, Cambridge, MA, 1996.
27. J. Sprave. Ein einheitliches Modell für Populationsstrukturen in evolutionären Algorithmen. PhD thesis, University of Dortmund, 1999.
28. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1998.
29. G. Warnecke. *Spanbildung bei metallischen Werkstoffen*. Technischer Verlag Resch, Gräfelfing, 1974.
30. K. Weinert, J. Mehnen, and G. Rudolph. Dynamic neighborhood structures in parallel evolution strategies. Technical Report CI-114/01, Collaborative Research Center 531, University of Dortmund, 2001.
31. K. Weinert and M. Stautner. Reconstruction of particle flow mechanisms with symbolic regression via genetic programming. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H. M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 1439–1443. Morgan Kaufmann, San Francisco, CA, 7–11 July 2001.
32. K. Weinert and M. Stautner. Reconstruction of physical correlations using symbolic regressions. Technical Report CI-116/01, Collaborative Research Center 531, University of Dortmund, 2001.
33. K. Weinert and M. Stautner. A new view on symbolic regression. In *Proceedings of the Fifth European Conference on Genetic Programming (EuroGP-2002)*, Kinsale, Ireland, April 3–5 2002. (Accepted).
34. K. Weinert, T. Surmann, and J. Mehnen. Evolutionary surface reconstruction using CSG-NURBS-hybrids. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H. M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, page 1456. Morgan Kaufmann, San Francisco, CA, 7–11 July 2001.

35. K. Weinert, T. Surmann, and J. Mehnen. Parallel surface reconstruction. In *Proceedings of the Fifth European Conference on Genetic Programming (EuroGP-2002)*, Kinsale, Ireland, 3–5 April 2002. (Accepted).
36. K. Weinert and A. Zabel. Modelling chip-building in orthogonal cutting by using a cellular automata/genetic programming approach. In C. Fye, editor, *Proceedings of the Second ICSC Symposium on Engineering of Intelligent Systems EIS 2000*, University of Paisley, Scotland, June 27–30 2000. ICSC International Computer Science Conference.
37. J. Q. Xie, A. E. Bayoumi, and H. M. Zbib. Fea modeling and simulation of shear localized chip formation in metal cutting. *International Journal of Machine Tools and Manufacture*, 38:1057–1087, 1998.