Reducing Bloat in GP with Multiple Objectives

Stefan Bleuler, Johannes Bader, and Eckart Zitzler

Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Switzerland
[bleuler, bader, zitzler]@tik.ee.ethz.ch

Summary. This chapter investigates the use of multiobjective techniques in genetic programming (GP) in order to evolve compact programs and to reduce the effects caused by bloating. The underlying approach considers the program size as a second, independent objective besides program functionality, and several studies have found this concept to be successful in reducing bloat. Based on one specific algorithm, we demonstrate the principle of multiobjective GP and show how to apply Pareto-based strategies to GP. This approach outperforms four classical strategies to reduce bloat with regard to both convergence speed and size of the produced programs on an even-parity problem. Additionally, we investigate the question of why the Pareto-based strategies can be more effective in reducing bloat than alternative strategies on several test problems. The analysis falsifies the hypothesis that the small but less functional individuals that are kept in the population act as building blocks for larger correct solutions. This leads to the conclusion that the advantages are probably due to the increased diversity in the population.

1 Motivation

The tendency of trees to grow rapidly during a genetic programming (GP) run is well known [16, 26, 2, 6] and may be explained by:

- The bigger trees get, the more code they contain that does not influence the fitness of the individuals. These so-called introns protect the individuals against the destructive effects of the crossover and mutation operators.
- The probability of finding a big tree that achieves a high fitness is greater than of finding a short program with the same behaviour (fitness-causes-bloat theory [19]).
- Removing bigger subtrees is much more likely to destroy the program than removing shorter ones, which leads to a bias for the preservation of long programs (removal-bias theory [25]).

This phenomenon, which is denoted as *bloating*, leads to several problems:

- Trees can grow quadratically [18]; this leads not only to excessive use of CPU time and memory but also makes the evaluation of trees infeasible.
- Smaller solutions usually generalize the training data better than bigger ones [2].
- When trees start to grow rapidly, so does the fraction of the tree constituted of introns. The recombination of individuals therefore usually comprises an exchange of introns, and the fitness of the population does not improve anymore; the GP run stagnates with high probability [2]. Moreover, when the system is bloating, the recombination of individuals may have no effect since introns will be usually exchanged, and this may lead to stagnation.

Therefore, normally at least, an upper limit for the program size is set manually. Several other strategies have been developed to address the problem of bloating, which can roughly be divided into two classes:

- Methods that modify the program structure and/or the genetic operators in order to remove or reduce the factors that cause bloat. Some examples are Automatically Defined Functions (ADFs) [17], Explicitly Defined Introns (EDIs) [2] and Deleting Crossover [5].
- Techniques that incorporate the program size as an additional factor in the selection process, e.g., as a constraint (size limitation) or as a penalty term (Parsimony Pressure [26]).

When bloating occurs, combinations of the different approaches are possible. Nevertheless, both types have certain disadvantages. For methods of the first class, e.g., ADF, EDI or Deleting Crossover, usually knowledge of how the program structure and the genetic operators interact with the effect of bloating is required. A difficulty with some methods of the second class is to optimally set the parameters associated with them, e.g., by choosing an appropriate parsimony factor when applying Constant Parsimony Pressure [26].

Pareto-based methods belong to the second class and have two advantages: They do not rely on problem knowledge and they do not require additional parameters to be set. The idea is to consider the program size as a second, independent objective besides program functionality and apply a Pareto-based method to the resulting bi-objective problem. The algorithm will then always prefer the smaller of two equally performing programs. As an additional side effect, both small but less functional and large but more complete programs will be kept in the population during the evolution. This basic strategy has proved to successfully reduce bloat in several studies [4, 9, 7, 11, 21, 3, 15]. However, it is still an open question why keeping many non-functional small individuals in the population helps in finding small and correct solutions quickly rather than distracting the search. A potential explanation is the increased diversity in the population. Alternatively, small individuals may be partial solutions to the problem, which can be combined by recombination into compact full solutions, thus acting as building blocks.

Using the method proposed in [4] as an example, the present chapter (i) describes how Pareto-based optimization methods can be applied to reduce code growth in GP and (ii) investigates what mechanisms make these methods effective. In the course of this chapter, we briefly discuss various traditional methods against bloat (Section 2.1), give an overview of alternative multiobjective approaches to fight bloat (Section 2.2), describe a particular method in detail (Section 3) and compare it to traditional techniques (Section 3.2), and finally investigate possible reasons for its effectiveness (Section 4).

2 Overview of Existing Approaches

2.1 Traditional Approaches to Reduce Bloat

Towards the end of a GP run introns grow rapidly and comprise almost all of the code while the optimization process stagnates (no fitness improvement anymore) [2]. Thus, the question is why simulated evolution favours programs with large sections of non-functional code over smaller solutions.

One explanation is that GP crossover is inhomogeneous, i.e., it does not exchange code fragments that have the same functionality in both parents. Therefore, crossover most often reduces the fitness of offspring relative to their parents by disrupting valuable code segments or placing them in a different context. Because crossover points are chosen randomly within an individual, the risk of disrupting blocks of functional code can be reduced substantially by adding introns. To keep this process from using too many machine resources, normally a limit on the tree depth or number of nodes is set manually; when an offspring individual exceeds this limit, one of its parents is added to the population instead. However, setting a reasonable limit is difficult. If the limit is too low, GP might not be able to find a solution. If it is too high, the evolution process will slow down because of the immense resource usage, and the chances of finding small solutions are very low. In the following, this setup will be named *Standard GP*. Here, the fitness F_i of individual i is defined as the error E_i of an individual's output compared to the correct solution

$$F_i = E_i,$$

where F_i is to be minimized.

Another obvious mechanism for limiting code size is to penalize larger programs by adding a size-dependent term to their fitness; this is called *Constant Parsimony Pressure* [5, 26]. The fitness of an individual i is calculated by adding the number of edges N_i, weighted with a parsimony factor α, to the regular fitness:

$$F_i = E_i + \alpha \cdot N_i$$

Soule and Foster [26] report that in some runs parsimony pressure drives the entire population to the minimum possible size. With a higher parsimony pressure the probability of a run suffering from this effect increases. This results in a lower probability of finding good solutions.

A third approach to tackling bloat is to optimize the functionality first and the size afterwards [12]. The formula for the fitness of an individual i depends on its own performance. An additional parameter ϵ comes into play; ϵ is the maximum acceptable error and can be set to zero for discrete problems. For fitness assignment the population is divided into two groups:

1. The individuals that have not yet reached an error equal to or smaller than ϵ get a fitness according to their error E_i without any pressure on the size:

$$F_i = E_i + 1 \text{ if } E_i > \epsilon$$

2. The fitness of individuals that have reached an error equal to or smaller than ϵ. The new fitness is calculated using the size N_i of individual i:

$$F_i = 1 - \tfrac{1}{N_i} \text{ if } E_i \leq \epsilon.$$

An individual with a large tree size will get a fitness near 1 while one with a large tree size will have a fitness closer to 0.

One advantage of this method is that the GP can find good solutions without being hampered since pressure on size is not applied until the individual has already reached the aspired-for performance. In runs where no acceptable solution is found, bloating will continue. Therefore it is useful to additionally set an upper limit on the tree size. In the following we will call this setup *Two Stage* for to the two stages of fitness evaluation.

Similar to this concept is a strategy called *Adaptive Parsimony Pressure*. Zhang and Mühlenbein have proposed an algorithm that varies the parsimony factor α during the optimization process [28]:

$$F_i(g) = E_i(g) + \alpha(g) \cdot C_i(g).$$

$C_i(g)$ stands for the complexity of individual i at generation g. The complexity can be defined in several ways [28], e.g., as the number of nodes in a tree or as normalized size obtained by dividing the individual's size by the maximum size in the population [5]. In contrast to the Two Stage strategy, the fitness function does not depend on the individual's performance but on the best performance in the population at generation g. The parsimony pressure used to calculate the fitness in generation g is increased substantially if the best individual in generation $g-1$ has reached an error below the threshold ϵ:

$$\alpha(g) = \begin{cases} \frac{1}{T^2} \cdot \frac{E_{best}(g-1)}{\hat{C}_{best}(g)} & \text{if } E_{best}(g-1) > \epsilon \\ \frac{1}{T^2} \cdot \frac{1}{E_{best}(g-1) \cdot \hat{C}_{best}(g)} & \text{otherwise.} \end{cases}$$

E_{best} is the error of the best performing individual in the population, T denotes the size of the training set and $\hat{C}_{best}(g)$ is the expected complexity of the best program in the next generation:

$$\hat{C}_{best}(g+1) = C_{best}(g) + \Delta C_{sum}(g),$$

where C_{best} stands for the complexity of the best performing individual in the population and $\Delta C_{sum}(g)$ is recursively defined as

$$\Delta C_{sum}(g) = \frac{1}{2} \left(C_{best}(g) - C_{best}(g-1) + \Delta C_{sum}(g-1) \right)$$

with the following starting value

$$\Delta C_{sum}(0) = 0.$$

The only parameter that has to be set manually is ϵ. Blickle [5] has reported results superior to those of Constant Parsimony Pressure when applying Adaptive Parsimony Pressure to a continuous regression problem, and equal results to those of Constant Parsimony Pressure when using it on a discrete problem.

In summary we can state that traditional methods aggregate the program function and program size in terms of one objective and fix a trade-off between these two criteria by means of a user-defined parameter.

2.2 Multiobjective Approaches

Using Size as Second Objective

Naturally, most optimization problems involve multiple, conflicting objectives which cannot be optimized simultaneously. This type of problem is often tackled by transforming the optimization criteria into a single objective which is then optimized using an appropriate single-objective method. The same is usually done when trying to address the phenomenon of bloat in GP by modifying the fitness evaluation or the selection process. Actually, there are two objectives: i) the functionality of a program and ii) the code size. While the second objective is traditionally converted into a constraint by limiting the size of a program, controlling the code size by adding a penalty term (Parsimony Pressure) corresponds to weighted-sum aggregation. Ranking the objectives, i.e., optimizing the functionality first and the size afterwards (Two Stage strategy), introduces a hierarchy on the objectives which in turn defines a total preorder on the search spaces.

Alternatively, Pareto-based methods can be applied by considering program functionality and program size as independent objectives. In this approach, small but functionally poor programs can coexist with large but good (in terms of functionality) programs, which in turn maintains population diversity during the entire run. It is important to note that only fitness assignment and selection is changed when switching from single-objective to multiobjective GP, while the other operators like mutation and recombination are not influenced. In 2001, three publications independently proposed the idea of using multiobjective methods for reducing bloat in GP, as first mentioned in [23] but not investigated in detail, and showed promising results [4, 9, 7]. In the following years additional studies have successfully used Pareto-based methods for bloat reduction [11, 21, 3, 15]. The remainder of this section summarizes these approaches and the key results of the respective studies. The method proposed in [4] serves as the basis for the analysis presented in this chapter.

Nondomination Tournament [9]

The authors propose a simple selection operator based on nondomination. In this scheme, the selection of one individual works as follows: A comparison set is randomly picked from the population and then candidate solutions are randomly chosen until one is found that is not dominated by any member of the comparison set; this individual is selected. To prevent the method from converging on small but non-functional programs an additional bias towards larger solutions is included in the domination criterion. Two different possibilities are compared: i) Using epsilon dominance on the program size, i.e., depending on the fitness f an individual i may dominate individual j even when it is larger than j ($f_i < f_j$ and $s_i < s_j + \varepsilon$). ii) Redefining the size objective such that it equals a threshold value for all trees that are smaller than this limit.

The approach is compared to standard GP on three symbolic regression problems and on the multiplexer problem. It is demonstrated that much smaller solutions can be found that are of similar quality. The number of fitness evaluations are similar, but due to the smaller average program size the running times are substantially smaller for the multiobjective method. Additionally, the preference for the different variants of the size bias changes with the problem.

FOCUS [7, 11]

Selection in the FOCUS algorithm works by discarding all weakly dominated individuals in the population. In order to promote diversity within the population, a diversity measure is used as a third objective in the optimization. However, unlike in most multiobjective EAs, diversity is measured in the parameter space. A distance measure for GP trees is defined and the average distance to the other members of the population is used as third objective function in the evaluation of a program.

This method is compared to standard GP on three instances of the parity bit problem. The experimental results show that some kind of diversity maintenance is necessary to keep the population from converging to small but non-functional trees. Including the diversity objective, FOCUS was, using fewer function evaluations, able to find correct solutions that are much smaller than those found by standard GP.

POPE-GP [3]

Another study uses NSGA-II [8] as selection operator in another overall method named POPE-GP [3]. Like all state-of-the-art evolutionary multiobjective optimization algorithms NSGA-II employs a diversity mechanism to distribute the individuals in the objective space. This eliminates the problem of convergence to small but non-functional programs. In an assessment on a classification problem, this approach yielded smaller programs with superior generalization compared to standard GP.

Biased Multiobjective Parsimony [21]

The authors argue that in most cases it is much easier for GP to find small non-functional programs than large correct ones, which is the reason why the evolution process can converge to small, non-functional individuals. To avoid this, a bias towards larger solutions is introduced. In contrast to the bias in [9], the user does not specify a target size but rather the relative importance of the two objectives. This is achieved by performing a tournament in which individuals are either compared based on their fitness (program functionality) only or based on their nondominated sorting rank [8]. By setting the probability p for using the fitness as criterion in the tournament, one can adjust the influence of the program size in selection, i.e., higher values of p lead to higher parsimony pressure.

The study uses four classical GP test problems (artificial ant, symbolic regression, multiplexer and even-parity) for comparing the proposed approach to standard GP and to two other single-objective strategies presented in the same publication. The results show that biased multiobjective parsimony is able to reduce the size of the solutions significantly for all problems in comparison to standard GP. But for higher parsimony pressure, which generates a highly significant reduction in program size, the fitness values start to increase compared to standard GP, i.e., the program functionality decreases. Consequently, the parsimony pressure must be carefully chosen to achieve a reduction while maintaining the quality of the solutions. The two single-objective strategies perform similarly to the multiobjective selection.

Other Related Approaches

The method presented in [15] is closely related to [9]. Here the selection consists of picking a random set of individuals and selecting all the nondominated solutions

from this set. Instead of replacing the selection operator with a multiobjective version as in most other approaches, Smits et al. [24] propose maintaining an external set of nondominated solutions and adapting the crossover operator to recombine one individual from the normal population with one individual from the nondominated set.

Methods like those presented in this section have successfully been used in several applications [27, 14, 29, 22]. And the same basic idea has found use in areas other than GP. In evolutionary design of classifiers, for example, the same problem with parsimony exists and the multiobjective methods presented for bloat in GP have effectively been applied [20, 10].

Summary of Multiobjective Approaches

Summarizing the studies discussed above, one can state that considering program size as a second objective beside program functionality and applying a Pareto-based optimization method has been highly successful in reducing bloat compared to standard GP. If a pure dominance-based fitness assignment scheme is used, this may lead to convergence to small, non-functional programs, thereby reducing the chance of finding a high-quality solution. The reason is that many more small programs exist than functional ones. Two basic strategies have been proposed to eliminate this problem: i) the introduction of a bias against small programs [9, 21], which gives rise to the difficult problem of correctly setting this bias or ii) the enforcement of diversity in the population with respect to either the parameter space [7, 11] or the objective space [4, 3].

3 A Multiobjective Approach to Reduce Bloat in Detail

This section describes how to apply a multiobjective approach to reducing bloat based on the example of the method presented in [4]. Additionally, it provides an empirical comparison of the multiobjective approach to four alternative strategies for reducing bloat.

3.1 Algorithm

The approach proposed in [4] uses an improved version of the Strength Pareto Evolutionary Algorithm (SPEA) for multiobjective optimization proposed in [32]. Besides the population, SPEA maintains an external set of individuals (archive) which contains the nondominated solutions among all solutions considered so far. The variant implemented here differs from the original SPEA only in the fitness assignment. In SPEA the fitness of an individual in the population depends on the "strengths" of the individual's dominators in the external set, but is independent of the number of solutions this individual dominates or is dominated by within the population. The potential problem arising with this scheme is illustrated in Figure 1. The Pareto-optimal front consists of only four solutions and the second dimension is highly discretized (as is the case for the application considered in Section 3.2, cf. Figure 11). As a consequence, the population is divided into four fitness classes, i.e., clusters which contain solutions having the same fitness. The fitness values only

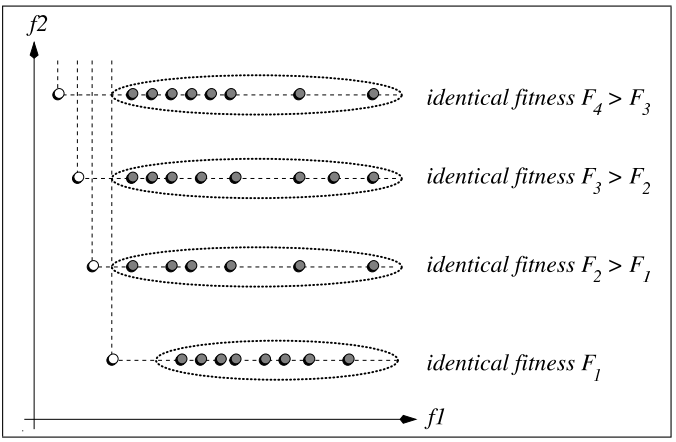

Fig. 1. A problematic situation with the original SPEA fitness assignment scheme in the case of a highly discretized objective space. The white points represent members of the external set while the grey points stand for individuals in the population

among clusters vary, not within clusters. Thereby the selection pressure towards the Pareto-optimal front is reduced substantially and may slow down the evolution process.

To avoid this situation, with the present algorithm both dominating and dominated solutions are taken into account for each individual. In detail, each individual i in the external set $\overline{P}$ *and* the population P is assigned a real value $S(i)$, its strength, representing the number of solutions it dominates:

$$S(i) = |\{j \mid j \in P + \overline{P} \wedge i \succeq j\}|,$$

where $|\cdot|$ denotes the cardinality of a set, $+$ stands for multiset union and the symbol $\succeq$ corresponds to the relation of weak Pareto dominance.[1] The strength of an individual is greater than or equal to 1 as each individual weakly dominates itself. Finally, the fitness $F(i)$ of individual i is calculated on the basis of the following formula:

$$F(i) = \sum_{j \succeq i} S(j).$$

That is, the fitness is determined by the strengths of its dominators. Note again that each individual weakly dominates itself and thus $F(i) \geq S(i)$. In contrast to SPEA, there is no distinction between members of the external set and population members.

It is important to note that fitness is to be minimized here, i.e., low fitness values correspond to high reproduction probabilities. The best fitness value is 1, which means that an individual is neither (weakly) dominated by any other individual nor (weakly) dominates another individual. A low fitness value is assigned to those individuals which

[1] A solution weakly dominates another solution if and only if it is not worse in any objective.

i) dominate only few individuals and
ii) are dominated by only few individuals (which in turn dominate only few individuals).

Therefore, not only is the search guided towards the Pareto-optimal front but also a niching mechanism is incorporated based on the concept of Pareto dominance. This enhances population diversity with respect to the objective space and successfully avoids convergence to small but non-functional programs, as will be demonstrated in Section 3.2.

For details of the SPEA implementation we refer the reader to [30]. The clustering procedure is not needed in this study because the size of the external set is unrestricted due to the small number of nondominated solutions emerging with the considered test problem.[2]

3.2 Experiments

In the following, we compare five methods — Standard GP, Constant Parsimony, Adaptive Parsimony, Two Stage and the SPEA variant — by evolving even-parity functions of different arities.

Methodology

The *even-parity* function was chosen because it is commonly used as a GP test problem [16, 26] and the complexity (arity = number of inputs) can be easily adapted to either the available machine resources or the performance of an algorithm. The Boolean *even-k-parity function* of k Boolean arguments returns `TRUE` if an even number of its Boolean arguments are `TRUE`, and otherwise returns `NIL`.

Parity functions are often used to check the accuracy of stored or transmitted binary data in computers because a change in the value of any one of its arguments toggles the value of the function. Because of this sensitivity to its inputs, the parity function is difficult to learn [17]. The training set consist of all 2^k possible input combinations. The error of an individual is measured as the number of input cases for which it did not provide the correct output value. A correct solution to the even-k-parity function is found when the error equals zero. We will call a run successful if it found at least one correct solution. For each setup 100 runs have been performed, and, in the following, usually the average values over 100 runs are reported. If not stated differently, the even-5-parity problem was used. Additionally, in a few runs even-parity functions of higher arities have been evolved.

Parameter Settings

After some test runs with Standard GP we decided to use a population size of 4,000 and a maximum of 200 generations; this setup performed best of all, keeping the product $Generations * Popsize = 800,000$ constant. All runs were processed up to generation 200, even if they found a correct program before generation 200. We

[2] The technique used here is a slight variation of the method later proposed under the name SPEA2 [31] which contains further improvements over SPEA.

set the initial depth for newly created trees to five and, in addition, restricted the maximum allowed depth of trees to 20, which is by far enough to generate correct solutions. It is important to note that only Standard GP and Two Stage runs (if no pressure is applied because no correct solution has been found) are affected by this limit. The other methods manage to keep the tree size so small that no significant part of the population reaches tree depths close to the limit.

The terminal set consists of all inputs $d_0, d_1, ..., d_{k-1}$ to the even-k-parity function. No numerical constants have been used. The function set consists of the following four Boolean functions: $\{AND, OR, IF, NOT\}$. Note that using the same function set without IF makes the task of evolving an even-parity function considerably more difficult. Preliminary tests for Constant Parsimony with different parsimony pressures of 0.001, 0.01, 0.1 and 0.2 showed the best results for $\alpha = 0.01$. This value has been used in all following Constant Parsimony runs.

For Adaptive Parsimony several settings from [5] have been used: The maximum acceptable error ϵ was set to 0.02. $E_i(g)$ was normalized with the maximum possible error. The best error that can be achieved is $E_i(g) = 0$. $C_i(g)$ was defined as the size $N_i(g)$ of an individual i normalized with the maximum size in population $N_{max}(g)$. In order to be able to use the formula given in Section 2 a constant $c = 0.01$ was added to the error measure.

Table 1 summarizes the parameters used for all runs (if not stated differently).

Table 1. Global parameter setting

Population size	4000
Generations	200
Maximum depth	$D_{max} = 20$
Maximum initial depth	$D_{initial} = 5$
Probability of crossover	$p_c = 0.9$
Probability of mutation	$p_m = 0.1$
Tournamentsize	$T = 7$
Reproduction method	Tournament
Function set	$\{AND, OR, IF, NOT\}$
Terminal set	$d_0, d_1, ..., d_{k-1}$
Constant Parsimony Pressure	$\alpha = 0.01$
Threshold (for Adaptive Pars.)	$\epsilon = 0.02$

Results

As expected, all methods have been able to find correct solutions in most of the 100 runs. Table 2 shows the percentage of successful runs, i.e., runs that found at least one correct solution within 200 generations. Two Stage and Standard GP have the same probability of solving the test problem since the fitness function is the same for both unless the concerned individual in Two Stage already represents a correct solution.

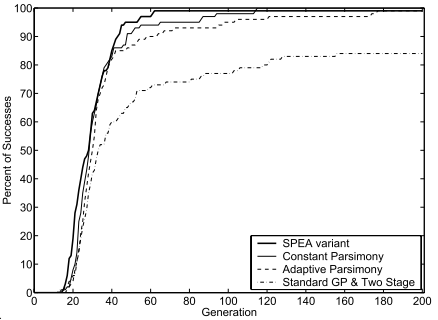

Fig. 2. Comparison of the success rates for the different methods relative to the generations. 100% means that all of the 100 runs found a solution before or in this generation

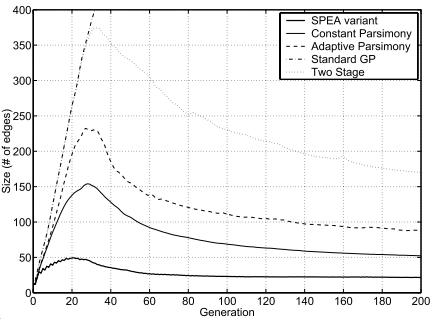

Fig. 3. Average tree size, mean of 100 runs per method

More information about how fast a method finds correct solutions can be obtained by calculating the probability of a run finding a correct solution within the first k generations. It is computed by summing over the runs that have found a correct solution by generation k. This probability is shown in Figure 2. It is interesting that all methods have found correct solutions before generation 20 in some runs. For all methods the probability of finding the first correct solution in the second half of the run is low. Increasing the arity of the even-parity function from 5 to 7 makes the problem much harder to solve. With an even-7-parity function, Standard GP did not produce one correct solution within 31 runs of 200 generations each. Parsimony was successful in ten and the SPEA variant in 22, out of 31 runs. This shows that keeping smaller trees in the population not only reduces the computational effort but also improves chances of solving the problem. For the even-9-parity function, the SPEA variant was successful within 500 generations in 17 out of 31 runs, and Constant Parsimony in 4 out of 31.

Table 2. Results compared for Standard GP, Two Stage, Constant Parsimony, Adaptive Parsimony and the SPEA variant

Method	Success Rate [%]	Smallest Av. Size	Mean Av. Size	Largest Av. Size
Standard GP	84	324.0	643.2	1701.8
Constant Pars.	100	26.2	52.3	106.9
Adaptive Pars.	99	23.0	87.1	714.9
Two Stage	84	25.7	170.1	867.6
SPEA variant	99	16.8	21.7	37.1

One of the main goals of reducing bloat is to keep the average tree size small in order to lower the computational effort required. Figure 3 shows the mean of average tree sizes in the population for 100 runs relative to the generation. Standard GP shows a rapid increase of average size until a significant part of the population reaches the maximum tree depth at about generation 20. From this point on, the increase in size gets slower. This is clearly an effect of limiting the tree depth. Out of ten runs where the tree depth was unlimited, none showed this saturation pattern. In contrast, tree size grew faster and faster, reaching an average size of 9,764 edges (average over 10 runs).

All of the other methods show common behaviour. After reaching a maximum between generation 20 and 30 the average size is reduced and stabilizes. Around the time when the average size reaches a maximum, the average error reaches a minimum. Maybe it is the general behaviour of algorithms that somehow favour small solutions, at least for discrete problems. An improvement in functionality is first achieved by a large individual and is followed by smaller programs with the same error. At the beginning of a run, when the average error is high, it is easy for evolution to improve functionality and the reduction of the average error is fast. The reduction in size mainly takes place when a lot of individuals have the same fitness. While fitness is changing fast this is not the case. Parsimony pressure with an α of 0.01, for example, mainly distinguishes between programs of equal performance. An individual may be 100 nodes larger than another and compensate for this with classifying only one additional test case correctly. Further investigations would be needed to justify the previously mentioned assumption.

Of more practical relevance is the fact that although the average size development shows a similar pattern for Two Stage, Constant Parsimony, Adaptive Parsimony and the SPEA variant the absolute values differ very much. As can be seen in Figure 3, the proposed SPEA variant has by far the smallest average size throughout the whole run. In generation 200 the average number of edges is down to 21.7; this is less than half of the second smallest average size which was attained by Constant Parsimony. Another important aspect is the range between the highest and the lowest final average size within all runs for one method. Table 2 lists the highest and the lowest final average size that occurred in 100 runs. For the SPEA variant the final average sizes vary only very little. At the other extreme is Two Stage. Some of the Two Stage runs never found a correct solution and therefore never experienced any pressure on tree size. These runs are exactly like Standard GP runs. Adaptive Parsimony performed considerably worse than Constant Parsimony (unlike in [5], where Adaptive Parsimony and Constant Parsimony achieved equal performance), and its final average sizes fell into a large range.

The second main goal when using methods against bloat is to retrieve compact solutions. The question is whether methods that keep the average tree size in the population low also produce small correct solutions. Figures 4 to 8 show a bar for each run. The height of the bar corresponds to the size of the smallest correct solution that was found during the whole run. If no correct solution was found there is no corresponding bar. For calculating the mean and median value only successful runs have been taken into account. It is shown that methods with low average tree sizes like the SPEA variant and Constant Parsimony were not only able to produce correct solutions but also found more compact solutions than methods with a larger average tree size. The average size of the smallest solutions for the SPEA variant is 21.1, which is close to the minimal possible tree size (17) for a solution to the

even-5-parity function using the given function set. This ideal solution was found in 22 runs. Every successful run found compact solutions; even the worst run found a solution of size 38. Although Constant Parsimony has a high probability of finding correct solutions within 200 generations, the size of the smallest solutions varies in a wide range. Once again the results of Adaptive Parsimony are worse than those of Constant Parsimony. Especially, the range of the sizes of the smallest solutions is larger with Adaptive Parsimony Pressure.

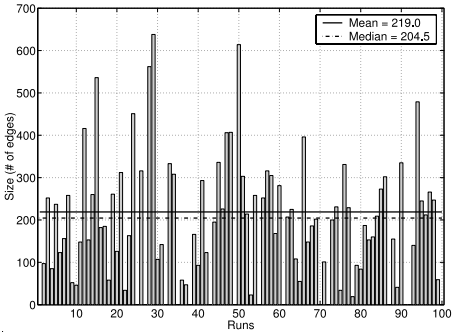

Fig. 4. Standard GP, size of the smallest correct solution

Some insight into why the SPEA variant is more successful than Constant Parsimony can be gained by looking at the distribution of the population in the (size, error)-plane. Figures 9 to 12 show the distribution of the population at generation 30 and 200 both for one representative run of the SPEA variant and one Constant Parsimony run. Each dot in the diagram represents one individual. The two runs for the SPEA variant and Constant Parsimony have been started with the same initial population. While the SPEA variant keeps a set of small individuals with different errors in the population during the whole run, Constant Parsimony moves the entire population towards lower errors and larger sizes. Around generation 30, when the average size reaches a maximum value and the average error a minimum value, parsimony pressure becomes effective and the population is moved back towards smaller sizes. The only small programs that are constantly kept in the population have an error of 16. Into this category also falls the smallest possible program that results from returning one input to the output. It is possible that in the variety of small trees that can be found in populations of the SPEA variant at all stages of the evolution, good building blocks for correct solutions are present.

4 Investigating the Mechanisms of Multiobjective Bloat Reduction

As demonstrated in the previous section and by all the studies described in Section 2.2, Pareto-based multiobjective optimization is successful at reducing bloat.

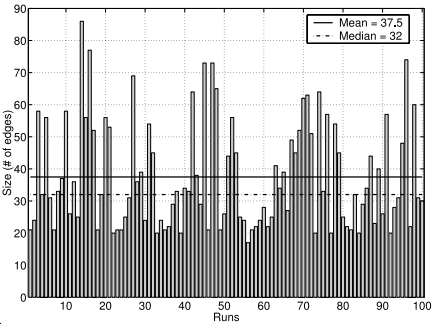

Fig. 5. Constant Parsimony, size of the smallest correct solution

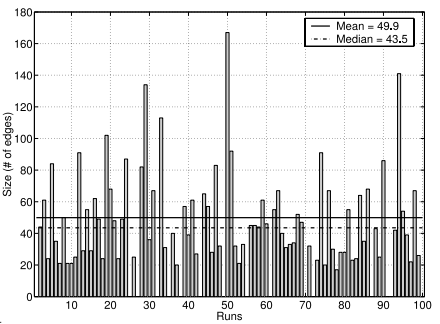

Fig. 6. Two Stage, size of the smallest correct solution

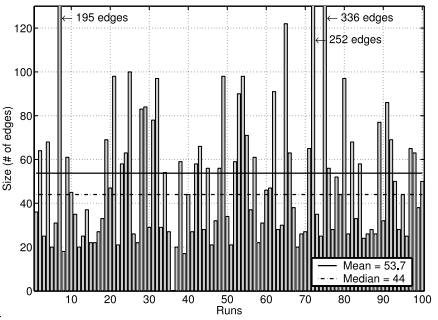

Fig. 7. Adaptive Parsimony, size of the smallest correct solution

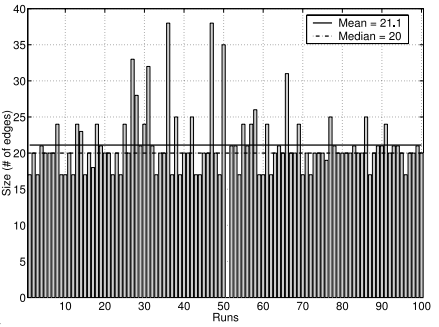

Fig. 8. SPEA variant, size of the smallest correct solution

While it is intuitive that maintaining a selection pressure towards smaller programs reduces bloat, it is not obvious why the multiobjective approach is particularly effective compared to alternative strategies. The algorithm used in Section 3 maintains a large portion of small non-functional programs in the population, as no preference for any of the two objectives size, and functionality is applied. This strategy seems to enhance the identification of a compact and correct solution rather than distracting the search algorithm as one would assume. In the following, a hypothesis concerning the cause of the observed behaviour will be presented and analysed.

4.1 Hypotheses

Figure 13 shows the hypothesized minimal-size solution found for the even-5-parity problem with the given operator set. This program is composed of subtrees that are themselves solutions to the parity bit problems for a lower number of inputs and that were often found by the multiobjective approach. This observation, together with the fact that even-parity programs can be obtained by programs for lower a number of bits, leads to the assumption that it was composed of solutions to subproblems by means of recombination. In this scenario, the multiobjective approach may support the existence of such small programs that are not correct solutions but that have relatively good functionality, as they are solutions to subproblems. So, the hypothesis

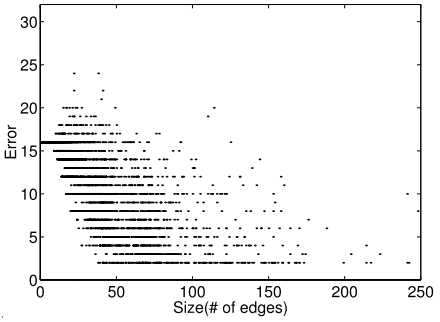

Fig. 9. SPEA variant population at generation 30

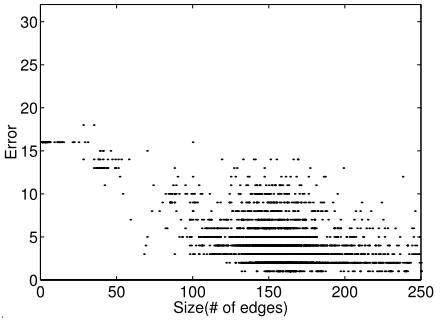

Fig. 10. Constant Parsimony population at generation 30

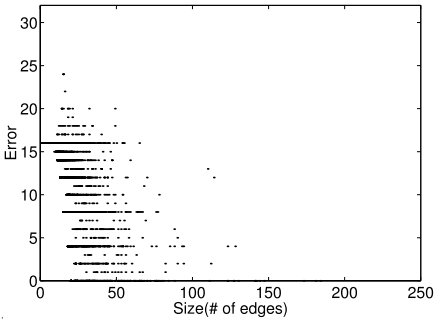

Fig. 11. SPEA variant population at generation 200

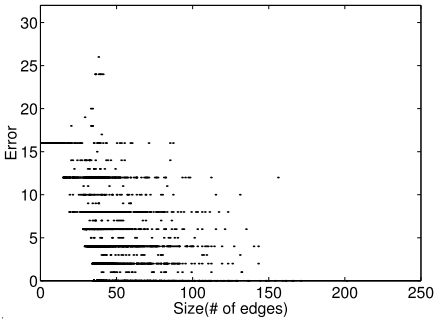

Fig. 12. Constant Parsimony population at generation 200

is that the small programs kept in the population act as building blocks for compact and correct solutions.

Alternatively, the advantage of the multiobjective optimization may be based on the increased genetic diversity. The existence of this effect was demonstrated in an empirical study in [1] where single-objective optimization problems were solved by adding objectives and applying a Pareto-based method. A closely related idea is that adding objectives makes a problem easier by removing local optima [13].

If the building block hypothesis describes the dominating factor leading to bloat reduction, the following can be expected to hold:

- The fitness assignment should be able to discriminate between small solutions that are solutions to subproblems or building blocks and random programs of the same size. If this is not the case, the EA will not prefer building blocks over random solutions.
- Through recombinations of solutions to subproblems, the functionality of offspring programs should often be largely better than the parents' functionality.
- Switching off recombination should strongly reduce the effectiveness of the multiobjective method.

In the following, the experimental results on multiple test problems for these effects will be analysed.

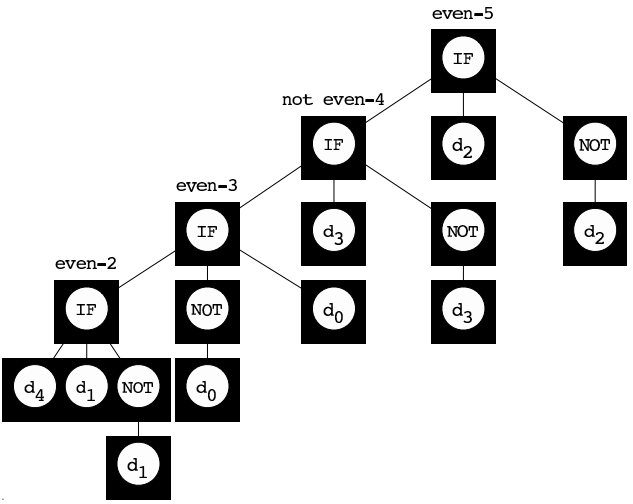

Fig. 13. Hypothesized minimal size solution found for the even-5-parity problem. The marked subtrees are solutions to parity problems of lower arities

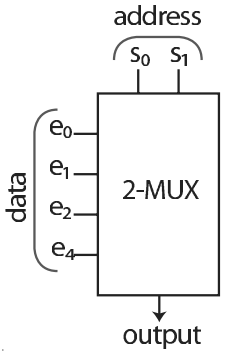

Fig. 14. Multiplexer problem ($k = 6$)

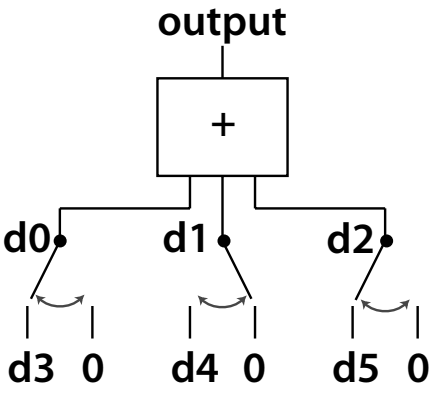

Fig. 15. Adder problem ($k = 6$)

4.2 Test Problems

Besides the parity bit problem described in Section 3.2 the following test problems are used for the analysis.

k-Multiplexer

A multiplexer is a device to select one of several data inputs and forward it to the output, cf. Figure 14. The k-multiplexer problem has m binary control inputs and n binary data inputs, where $n = 2^m$ and $k = m + n$. Thus, the feasible values for k are $k = \{3, 6, 11, 20, 37, \ldots\}$

k-Hamming Distance

Here, the task is to calculate the Hamming distance between the first half and the second half of the binary input string of length k. Obviously, k is restricted to being even. This test problem was specifically designed to allow a stepwise buildup of correct programs, since the solutions that calculate the correct Hamming distance for a part of the input string have a relatively high score on the complete problem.

k-Adder

A related test problem is the k-adder where the first $\frac{k}{2}$ bits of the k binary input d_i determine which of the remaining bits are added; cf. Figure 15. Thus, the output is calculated as follows:

$$a = \sum_{i=0}^{k} d_i \cdot d_{i+\frac{k}{2}}. \tag{1}$$

Operators

The same operators as for the parity bit have been used for the multiplexer problem, namely NOT, OR, AND, and IF. For the Hamming distance and the adder an additional binary plus operator + was introduced.

4.3 Results

This section tests the building block hypothesis described above by analysing experimental results for the different effects that should be observed if the hypothesis holds.

Fitness Discrimination of Small Programs

For the population to contain a significant number of small programs that are solutions to subproblems, they must exhibit better fitness values than random programs of the same size, e.g., when scored on the even-5-parity problem, a solution to the even-3-parity problem should be preferred to a random program of the same size. One characteristic of solutions to subproblems is that they do not use all available inputs. Accordingly, the possible fitness ranges for programs that do not use all of the provided inputs (as do solutions to subproblems) are plotted in Figures 16–19. On the k-parity problem, all solutions that use less than k inputs have equal fitness as they provide the correct result to exactly half of the test cases. Thus, solutions to subproblems or other good building blocks for compact full solutions cannot prevail in the population despite the significant portion of small programs maintained by the Pareto-based selection. Nevertheless, the multiobjective method was highly successful on the parity problems. This gives a first indication against the building block hypothesis. For the other test problems the same problem does not appear, and the Hamming distance problem and the adder problem have been specifically designed such that solutions to subproblems score relatively well on the full problem.

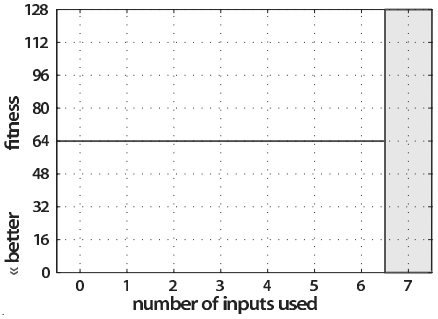

Fig. 16. Fitness ranges for small programs on the even-7-parity problem

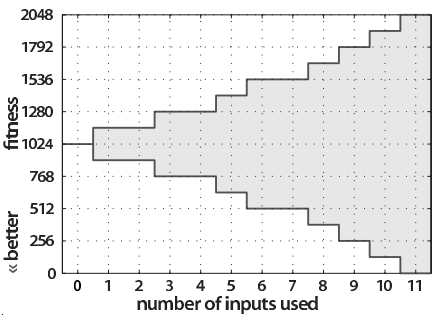

Fig. 17. Fitness ranges for small programs on the 11-multiplexer problem

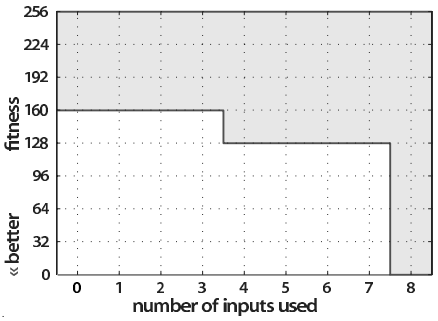

Fig. 18. Fitness ranges for small programs on the 8-Hamming distance problem

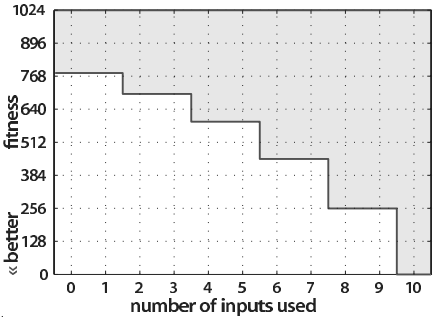

Fig. 19. Fitness ranges for small programs on the 10-adder problem

Fitness Changes in Recombination

If the crossover operator successfully combines solutions to subproblems or other building blocks into a full solution, the fitness value of the offspring will be substantially better than the parents' fitness. If such recombinations are a major origin of good solutions, we can expect to often see large fitness increases from parents to offspring. Figures 20–22 show the fitness differences between one parent and one offspring appearing in all recombinations of 25 runs. Large changes in fitness are very rare. This indicates that recombination of building blocks into good programs is extremely rare on our test problems.

Effect of Single-Parent Variation

The hypothesis states that the multiobjective methods maintain more promising building blocks in the population than alternative methods like Constant Parsimony. Thus, its performance should depend more on recombination than that of constant parsimony. Consequently, using single-parent variation, i.e., switching off recombination, should affect the SPEA variant much more than Constant Parsimony. We have tested this on the even-7-parity problem using the parameter settings as described in Table 1, except for the tournament size in Constant Parsimony, which was

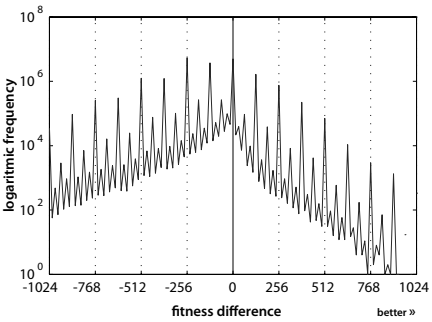

Fig. 20. Fitness differences in recombination on the 11-multiplexer problem

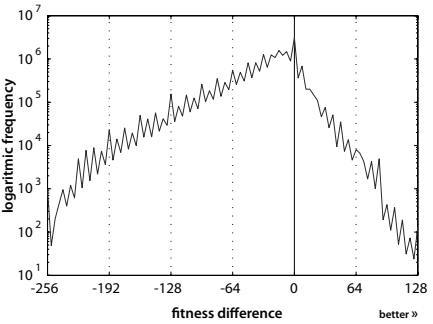

Fig. 21. Fitness differences in recombination on the 8-hamming problem

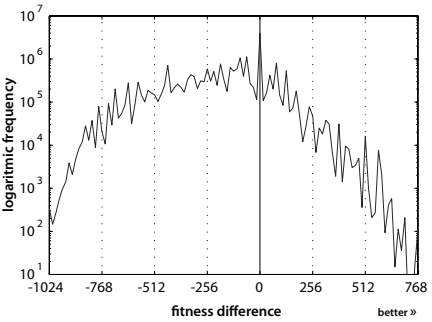

Fig. 22. Fitness differences in recombination on the 10-adder problem

set to 10. In order to compensate for the reduced variation without recombination, the mutation rate was increased to 0.9. Figures 23 and 24 show in how many of the 10 runs a correct solution was found for the three different settings standard ($p_c = 0.9$, $p_m = 0.1$), high mutation ($p_c = 0.9$, $p_m = 0.9$), and no crossover ($p_c = 0$, $p_m = 0.9$). For both methods the number of successful runs is similar and does not change heavily. Another performance indicator is the speed of convergence. Here, both algorithms take much longer to find the first correct solution, as shown in Figures 25 and 26 due to the increased mutation rate, but no significant difference in the influence of crossover exists. Lastly, we compare the sizes of the smallest correct solutions found by the two methods; cf. Figures 27 and 28. Again, the performance is influenced adversely by the increased mutation rate but there is no significant influence of the recombination. In summary, the performance of the SPEA variant is not more dependent on recombination than the performance of Constant Parsimony.

5 Summary

Bloating is a well known problem of variable-length representations, as used in genetic programming, and various strategies have been proposed to address it. A recent

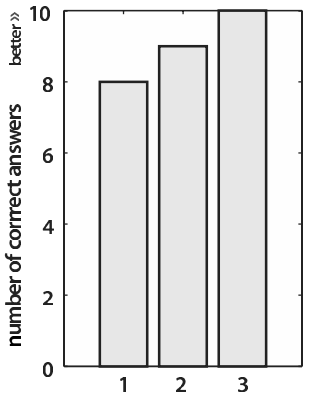

Fig. 23. Number of successful runs for the SPEA variant on the 7-even-parity problem. 1) standard settings, 2) high mutation rate, 3) high mutation rate and no recombination

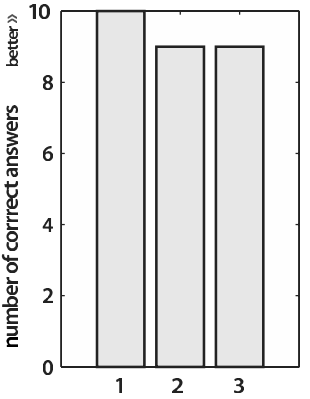

Fig. 24. Number of successful runs for the Constant Parsimony on the 7-even-parity problem. 1) standard settings, 2) high mutation rate, 3) high mutation rate and no recombination

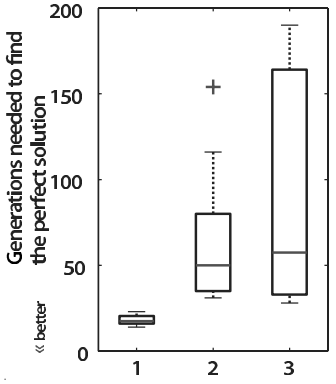

Fig. 25. Generation of the first correct solution for the SPEA variant on the 7-even-parity problem. 1) Standard settings, 2) high mutation rate, 3) high mutation rate and no recombination

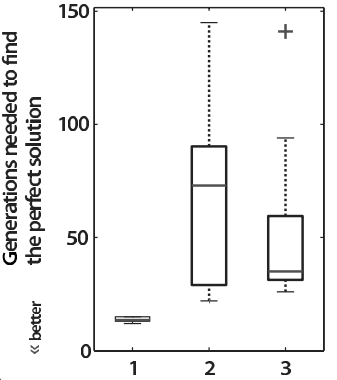

Fig. 26. Generation of the first correct solution for the Constant Parsimony on the 7-even-parity problem. 1) Standard settings, 2) high mutation rate, 3) high mutation rate and no recombination

development is to explicitly use the underlying objectives of program functionality and program size in a multiobjective optimization method. Several variants of this approach have been proposed, all of which successfully reduced code growth compared to standard GP with depth limitation, on a variety of discrete and continuous test problems. Here, we have discussed how to apply Pareto-based multiobjective methods to the problem of bloat on the example of the SPEA variant published in [4]. The experimental validation on the parity-bit test problem showed that this method not only reduces code growth compared to standard GP but also outperforms three alternative methods for bloat control with respect to average size of

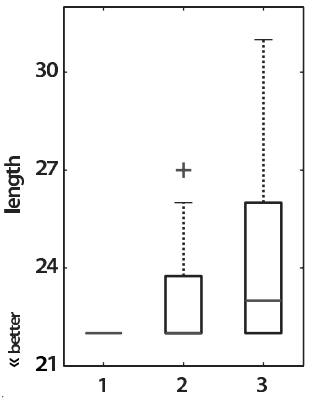

Fig. 27. Size of the smallest correct solution found in each run of the SPEA variant on the 7-even-parity problem. 1) Standard settings, 2) high mutation rate, 3) high mutation rate and no recombination

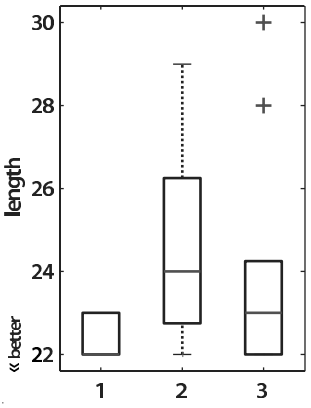

Fig. 28. Size of the smallest correct solution found in each run of Constant Parsimony on the 7-even-parity problem. 1) Standard settings, 2) high mutation rate, 3) high mutation rate and no recombination

the programs, which is decisive for the overall computational effort, the size of the smallest correct solutions and the best program functionality.

Additionally, we tried to identify the cause for these improvements as it is not obvious why keeping small and non-functional programs in the population can improve the quality of the results rather than distracting the search. We have formulated the hypothesis that small programs may act as building blocks for compact correct solutions. These building blocks could then be combined into compact correct solutions by recombination, whereas alternative methods which do not keep small but non-functional programs in their populations cannot profit from this effect. Several tests revealed evidence against this hypothesis. In particular,

- the multiobjective method is also successful when the fitness of small solutions that may act as building blocks cannot be distinguished from the fitness of random programs of the same size,
- recombination does very rarely leads to the large improvements in fitness that would be expected for successful combinations of building blocks, and
- switching off recombination does not seem to influence the capabilities of the multiobjective approach.

Therefore, we conclude that the positive effects of maintaining small but non-functional programs in the population are mainly due to increased genetic diversity, as described in [1], or the closely related concept of changes in the fitness landscape which induce a lower number of local optima [13]. Additional experiments will be necessary to further verify these conclusions.

One explanation of why maintaining diversity is important with respect to small trees is that the recombination gets more effective the smaller the trees are (as a comparison between Figures 9 and 10 reveals).

References

[1] H. A. Abbass and K. Deb. Searching under Multi-evolutionary Pressures. In C. M. Fonseca et al., editors, *Evolutionary Multi-Criterion Optimization. Second International Conference, EMO 2003*, pages 391–404, Berlin, Germany, 2003. Springer. Lecture Notes in Computer Science. Volume 2632.

[2] W. Banzhaf, F. D. Francone, R. E. Keller, and P. Nordin. *Genetic Programming: An Introduction.* Morgan Kaufmann, San Francisco, CA, 1998.

[3] Y. Bemstein, X. Li, V. Ciesielski, and A. Song. Multiobjective parsimony enforcement for superior generalisation performance. In IEEE, editor, *CEC 04*, pages 83–89, 2004.

[4] S. Bleuler, M. Brack, L. Thiele, and E. Zitzler. Multiobjective Genetic Programming: Reducing Bloat by Using SPEA2. In *Congress on Evolutionary Computation (CEC-2001)*, pages 536–543, Piscataway, NJ, 2001. IEEE.

[5] T. Blickle. Evolving Compact Solutions in Genetic Programming: A Case Study. In H. M. Voigt et al., editors, *PPSN IV*, pages 564–573. Springer-Verlag, 1996.

[6] T. Blickle and L. Thiele. Genetic programming and redundancy. In J. Hopf, editor, *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*, pages 33–38, 1994.

[7] E. D. De Jong, R. A. Watson, and J. B. Pollack. Reducing Bloat and Promoting Diversity using Multi-Objective Methods. In L. Spector et al., editors, *Genetic and Evolutionary Computation Conference (GECCO 2001)*, pages 11–18. Morgan Kaufmann Publishers, 2001.

[8] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In M. Schoenauer et al., editors, *Parallel Problem Solving from Nature (PPSN VI)*, Lecture Notes in Computer Science Vol. 1917, pages 849–858. Springer, 2000.

[9] A. Ekárt and S. Z. Németh. Selection Based on the Pareto Nondomination Criterion for Controlling Code Growth in Genetic Programming. *Genetic Programming and Evolvable Machines*, 2:61–73, 2001.

[10] A. Hunter. Expression Inference - Genetic Symbolic Classification Integrated with Non-linear Coefficient Optimisation. In *AISC 02*, LNCS. Springer, 2002.

[11] E. D. D. Jong and J. B. Pollack. Multi-objective methods for tree size control. *Genetic Programming and Evolvable Machines*, 4:211–233, 2003.

[12] T. Kalganova and J. F. Miller. Evolving More Efficient Digital Circuits by Allowing Circuit Layout Evolution and Multi-Objective Fitness. In A. Stoica et al., editors, *Proceedings of the 1st NASA/DoD Workshop on Evolvable Hardware (EH'99)*, pages 54–63, Piscataway, NJ, 1999, 1999. IEEE Computer Society Press.

[13] J. D. Knowles, R. A. Watson, and D. W. Corne. Reducing Local Optima in Single-Objective Problems by Multi-objectivization. In E. Zitzler et al., editors, *Evolutionary Multi-Criterion Optimization (EMO 2001)*, volume 1993 of *Lecture Notes in Computer Science*, pages 269–283, Berlin, 2001. Springer-Verlag.

[14] A. Kordon, E. Jordaan, L. Chew, G. Smits, T. Bruck, K. Haney, and A. Jenings. Biomass Inferential Sensor Based on Ensemble of Models Generated by Genetic Programming. In *GECCO 04*, LNCS, pages 1078–1089. Springer, 2004.

[15] M. Kotanchek, G. Smits, and E. Vladislavleva. Pursuing the Pareto Paradigm Tournaments, Algorithm Variations & Ordinal Optimization. In R. L. Riolo, T. Soule, and B. Worzel, editors, *Genetic Programming Theory and Practice IV*, volume 5 of *Genetic and Evolutionary Computation*, chapter 3. Springer, 2006.

[16] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[17] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, Massachusetts, 1994.

[18] W. B. Langdon. Quadratic Bloat in Genetic Programming. In D. Whitley et al., editors, *GECCO 2000*, pages 451–458, Las Vegas, Nevada, USA, 10-12 2000. Morgan Kaufmann. ISBN 1-55860-708-0.

[19] W. B. Langdon and R. Poli. Fitness Causes Bloat. In P. K. Chawdhry et al., editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22, Godalming, GU7 3DJ, UK, 1997. Springer-Verlag.

[20] X. Llorà, D. E. Goldberg, I. Traus, and E. Bernadó. Accuracy, parsimony, and generality in evolutionary learning systems via multiobjective selection. In *Learning Classifier Systems*, pages 118–142. Springer. Lecture Notes in Artificial Intelligence Vol. 2661, 2002.

[21] L. Panait and S. Luke. Alternative Bloat Control Methods. In *GECCO 04*, LNCS, pages 630–641. Springer, 2004.

[22] D. Parrot, L. Xiandong, and V. Ciesielski. Multi-objective techniques in genetic programming for evolving classifiers. In *CEC 05*, pages 1141–1148. IEEE, 2005.

[23] K. Rodríguez-Vázquez, C. M. Fonseca, and P. J. Fleming. Multiobjective genetic programming: A nonlinear system identification application. In J. R. Koza, editor, *Late Breaking Papers at the 1997 Genetic Programming Conference*, pages 207–212, Stanford University, CA, USA, 13–16 1997. Stanford Bookstore. ISBN 0-18-206995-8.

[24] G. F. Smits and M. Kotanchek. Pareto-Front Exploitation in Symbolic Regression. In *Genetic Programming Theory and Practice II*, volume 8. Springer, 2005.

[25] T. Soule and J. A. Foster. Removal Bias: a New Cause of Code Growth in Tree Based Evolutionary Programming. In *1998 IEEE International Conference on Evolutionary Computation*, pages 781–186, Anchorage, Alaska, USA, 1998. IEEE Press. URL `http://citeseer.ist.psu.edu/313655.html`.

[26] T. Soule and J. A. Foster. Effects of Code Growth and Parsimony Pressure on Populations in Genetic Programming. *Evoluationary Computation*, 6(4): 293–309, 1999.

[27] M. Streeter and L. A. Becker. Automated Discovery of Numerical Approximation Formulae via Genetic Programming. *Genetic Programming and Evolvable Machines*, 4(3):255–286, 2003.

[28] B.-T. Zhang and H. Mühlenbein. Balancing Accuracy and Parsimony in Genetic Programming. *Evoluationary Computation*, 3(1):17–38, 1995.

[29] Y. Zhang and P. I. Rockett. Evolving optimal feature extraction using multiobjective genetic programming: a methodology and preliminary study on edge detection. In *GECCO 05*, pages 795–802, New York, NY, USA, 2005. ACM Press.

[30] E. Zitzler. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. PhD thesis, Swiss Federal Institute of Technology (ETH) Zürich, Switzerland, 1999.

[31] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization. In K. Giannakoglou et al., editors, *Evolutionary Methods for Design, Optimisation and Control with Application to Industrial Problems (EUROGEN 2001)*, pages 95–100. International Center for Numerical Methods in Engineering (CIMNE), 2002.

[32] E. Zitzler and L. Thiele. Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.