Proceedings of the 2001 IEEE Congress on Evolutionary Computation Seoul, Korea • May 27-30, 2001

Multiobjective Genetic Programming: Reducing Bloat Using SPEA2

Stefan Bleuler, Martin Brack, Lothar Thiele and Eckart Zitzler
Computer Engineering and Communication Networks Lab (TIK)
Swiss Federal Institute of Technology (ETH)
Gloriastr. 35
CH–8092 Zürich
Switzerland
{sbleuler / mbrack}@ee.ethz.ch, {thiele / zitzler}@tik.ee.ethz.ch

Abstract- This study 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 proposed approach considers the program size as a second, independent objective besides the program functionality. In combination with a recent multiobjective evolutionary technique, SPEA2, this method outperforms four other strategies to reduce bloat with regard to both convergence speed and size of the produced programs on a even-parity problem.

1 Introduction

The tendency of tree sizes to grow rapidly during a Genetic Programming (GP) run is well known [Koz92, SF99, BFKN98, BT94]. There are several reasons why it is useful to take some measures against this phenomenon of *bloating*:

- The excessive use of CPU time and memory.
- Smaller solutions generalize the training data better than bigger ones [BFKN98].
- When tree sizes start to grow rapidly a GP run almost always stagnates. The fitness of the population is not improving anymore [BFKN98].

For these reasons 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 (ADF) [Koz94], Explicitly Defined Introns (EDI) [BFKN98] and Deleting Crossover [Bli96].
- Techniques that incorporate the program size as an additional, but hidden objective, e.g., as a constraint (size limitation), as a penalty term (Parsimony Pressure [SF99]), etc.

Combinations of different approaches are possible. Nevertheless, both types have certain disadvantages. For methods of the first class usually knowledge on how the program structure and the genetic operators interact with the effect of bloating is required. A difficulty with the second class of methods is to optimally set the parameters associated with them, e.g., choosing an appropriate parsimony factor when applying Constant Parsimony Pressure [SF99].

With multiobjective optimization algorithms it is possible to optimize towards several objectives at the same time by searching the so-called Pareto-optimal solutions. There are only a few studies which perform a multiobjective optimization in the context of GP. For instance, in [Lan96] data structures were evolved by treating the different operations of a list problem as separate objectives. In contrast to these studies, we here pursue the idea of reducing bloat by introducing the program size as a second, independent objective besides the program functionality. As we will show in the remainder of the paper, this approach in combination with a particular multiobjective optimization algorithm, SPEA2, is able to find more compact programs in fewer generations than existing approaches of this class (explicit incorporation of the program size) such as Parsimony Pressure on the even-parity problem.

This paper is organized as follows. Some background information about bloating and existing methods used to reduce bloating is given in the Section 2. Afterwards, we discuss the motivation for our approach and present some arguments why this approach is promising. The multiobjective optimization procedure, SPEA2, which forms the basis for our investigation is briefly sketched in Section 4, and Section 5 describes the experiments results where SPEA2 is compared with four other methods to reduce code growth. Finally, our conclusions and potential future research directions are the subject of Section 6.

2 Related Work

Several studies have examined possible reasons for bloating [LP97, BT94, BFKN98]. The increase in code size is an effect of so-called *introns*, parts of the tree that do not affect the individual's functionality. 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) [BFKN98]. Thus, the question is why evolution favors programs with large section of non-functional code over smaller solutions. One explanation is that GP crossover is inhomologous, 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 hinder this effect from using too much machine resources normally a limit on tree depth or number of nodes is set manually. 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 evolution will slow down because of the immense resource usage and 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.

$$F_i = E_i$$

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* [Bli96, SF99]. 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 [SF99] report that in some runs Parsimony Pressure drives the entire population to the minimal possible size. With a higher parsimony pressure the probability of a run to suffer from this effect is increasing. This results in a lower probability of finding good solutions.

Another alternative is to optimize the functionality first and the size afterwards [KM99]. The formula for the fitness of an individual i depends on its own performance. It is necessary to set a maximal acceptable error ϵ. For discrete problems ϵ can be set to zero. 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:

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

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

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

An individual with a large tree size will get a fitness near one while a small one will be much closer to zero.

One advantage of this method is that pressure on size will not hinder GP from finding good solutions because no pressure is applied unless the individual has already reached the aspired performance. In runs where no acceptable solution is found bloating will continue. Therefore it is useful to additionally set an upper limit on tree size. In the following we will call this setup *Two Stage* according to the two stages of fitness evaluation.

Similar to this is a strategy called *Adaptive Parsimony Pressure*. Zhang and Mühlenbein have proposed an algorithm that varies the parsimony factor α during the evolution [ZM95]:

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

$C_i(g)$ stands for the complexity of individual i at generation g. The complexity can be defined in several ways [ZM95]. For instance as the number of nodes in a tree or as normalized size by dividing the individual's size by the maximum size in population [Bli96]. 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 the 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. $\hat{C}_{best}(g)$ is an estimation of the complexity of the best program, estimated at generation $(g-1)$ it is used to normalize the influence of the parsimony pressure. C_{best} stands for the complexity of the best performing individual in the population.

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

with a recursively defined $\Delta C_{sum}(g)$

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

and the following starting value

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

The only parameter that has to be set manually is ϵ. Blickle [Bli96] has reported superior results compared to Constant Parsimony Pressure when applying Adaptive Parsimony Pressure to a continuous regression problem and equal results as with Constant Parsimony Pressure when using it on a discrete problem.

3 Multiobjective Optimization: Tree Size as a 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), is slightly different, but still requires the incorporation of preference information as with the other techniques.

Alternatively, there exist methods which treat all objectives equally. Instead of restricting the search *a priori* to one solution as with the aforementioned strategies, they try to find or approximate the so-called *Pareto-optimal set* consisting of solutions which cannot be improved in one objective without degradation in another. In the last decade several evolutionary algorithms (EAs) have been developed for this optimization scenario, and some studies [ZT99, ZDT00] showed for a number of test problems that this type can be superior to, e.g., weighted-sum aggregation in terms of computational effort and quality of the solutions found (when an elitist EA is used). This was the motivation for applying a multiobjective EA to the problem of bloat in GP by considering program functionality and program size as independent objectives. In this approach, small, but functionally poor program can coexist with large, but good (in terms of functionality) programs, which in turn maintains population diversity during the entire run. We will give evidence for our assumption that thereby more compact programs can be found in fewer generations in Section 5.

4 SPEA2 for Multiobjective Optimization

In this paper we use an improved version of the Strength Pareto Evolutionary Algorithm (SPEA) for multiobjective optimization proposed in [ZT99]. Besides the population SPEA maintains an external set of individuals (archive) which contains the nondominated solutions among all solutions considered so far. In each generation the external set is updated and if necessary pruned by means of a clustering procedure. Afterwards, individuals in population and external set are evaluated interdependently, such that external set members have better fitness values than the population members. Finally, selection is performed on the union of population and external set and recombination and mutation operators are applied as usual. As SPEA has shown very good performance in different comparative studies [ZT99, ZDT00], it has been a point of reference in various recent investigations, e.g., [CKO00]. Furthermore, it has been used in different applications, e.g., [LMBoZ01].

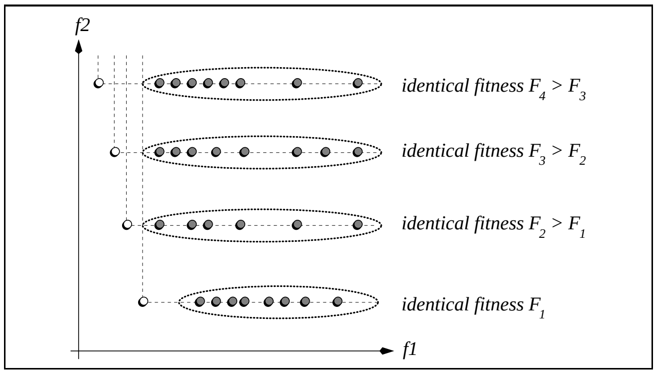
f2
f1
identical fitness F4 > F3
identical fitness F3 > F2
identical fitness F2 > F1
identical fitness F1

Figure 1: Illustration of SPEA's fitness assignment scheme in the case of a highly discretized objective space. The white points represent members of the external set while the gray points stand for individuals in the population.

SPEA2, which incorporates a close-grained fitness assignment strategy and an adjustable elitism scheme, is described in [ZLT01]. 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 it is the case for the application considered in Section 5, cf. Figure 11). As a consequence, the population is divided into four fitness classes, i.e., clusters which contain solutions having the same fitness. Only the fitness values among clusters vary, but not within the 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 SPEA2 for each individual both dominating and dominated solutions are taken into account. 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 or equal one 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

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

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 one, 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

i) dominate only few individuals and

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

Thereby, not only the search is guided towards the Pareto-optimal front but also a niching mechanism is incorporated which is based on the concept of Pareto dominance.

For details of the SPEA implementation we refer to [Zit99]. 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.

5 Experiments

We compared the following five methods: Standard GP, Constant Parsimony, Adaptive Parsimony, Two Stage and SPEA2 by evolving even-parity functions of different arities.

5.1 Methodology

The *even-parity* function was chosen because it is commonly used as a GP test problem [Koz92, SF99] 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 [Koz94]. 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. Given values are therefore normally averaged over 100 runs. 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.

5.2 Parameter Settings

After some test runs with Standard GP we decided to use a population size of 4000 and maximum number of 200 generations, this setup performed best of all that have been used by keeping the product $Generations * Popsize = 800000$ constant. All runs were processed up to generation 200 also if they found a correct program before generation 200. We set the initial depth for newly created trees to 5 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 executed 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 [Bli96] have been used: The maximal acceptable error ϵ was set to 0.02. $E_i(g)$ was normalized with the maximal 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.)	$\epsilon = 0.02$

5.3 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.

Table 2: Results compared for Standard GP, Two Stage, Constant Parsimony, Adaptive Parsimony and SPEA2.

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
SPEA2	99	16.8	21.7	37.1

More information about how fast a method finds correct solutions can be shown by calculating the probability of a run to find a correct solution within the first k generations. It is attained by adding up the number of runs out of a total of 100 that have found a correct solution by generation k. This probability is shown in Figure 2. Interesting is, 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 even-7-parity function Standard GP did not produce one correct solution within 31 runs of 200 generations each. Parsimony was successful in 10 and SPEA2 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 SPEA2 was successful within 500 generations in 17 out of 31 runs and Constant Parsimony in 4 out of 31. If Constant Parsimony would benefit from setting another α for a higher arity is unclear. As with SPEA2 we wanted see the performance on a higher arity without any parameter change.

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 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 of size is getting 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 contrary tree size grew faster and faster reaching an average size of 9764 edges (average over 10 runs).

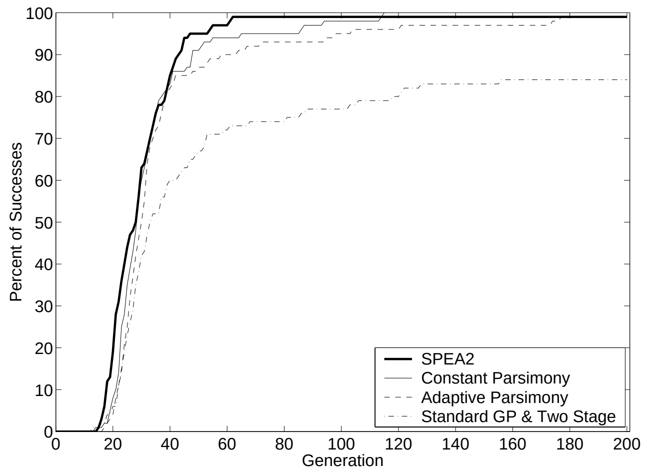

Figure 2: Comparison of the success rates for the different methods relative to the generations. 100% means that all of the 100 runs found a solution before or in this generation.

All of the other methods show a common behavior. 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. We assume that it is the general behavior of algorithms that somehow favor 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. A individual may be 100 nodes larger than another and compensate this with only classifying one additional test case correctly. Further investigations would be needed to justify the abovementioned 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 SPEA2 the absolute values differ very much. As can be seen in Figure 3 SPEA2 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 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 SPEA2 the final average sizes vary only very little. On 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 and its final average sizes fell into a large range. We have not been able to get the good results reported in [Bli96] where equal performance of Adaptive Parsimony and Constant Parsimony has been found for a discrete problem .

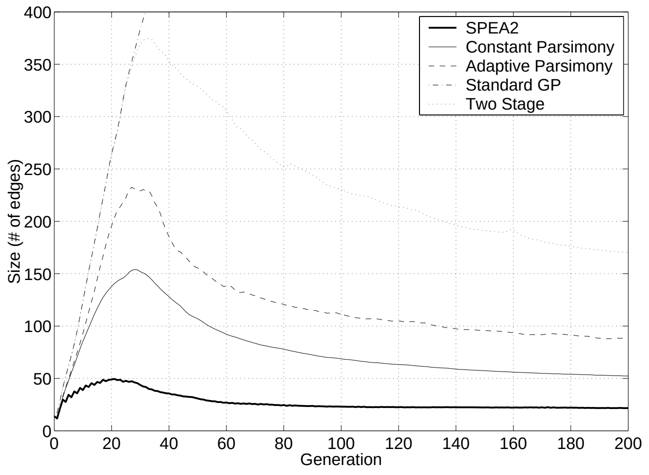

Figure 3: Average tree size, mean of 100 runs per method

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 SPEA2 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 SPEA2 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 as 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.

Some insight in why SPEA2 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 SPEA2 run and one Constant Parsimony run. Each dot in the diagram represents one indi-

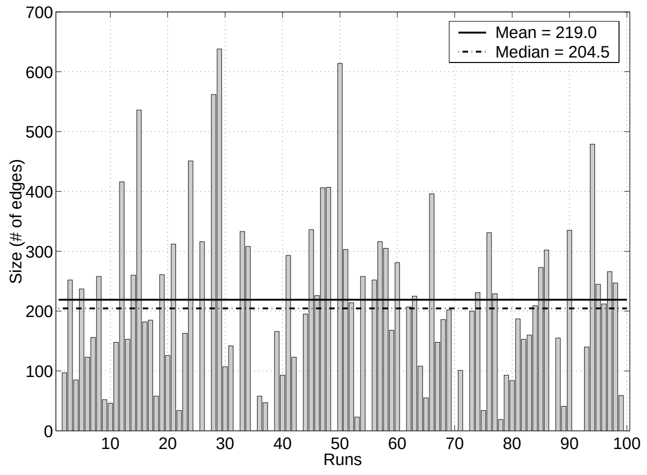

Figure 4: Standard GP, size of the smallest correct solution.

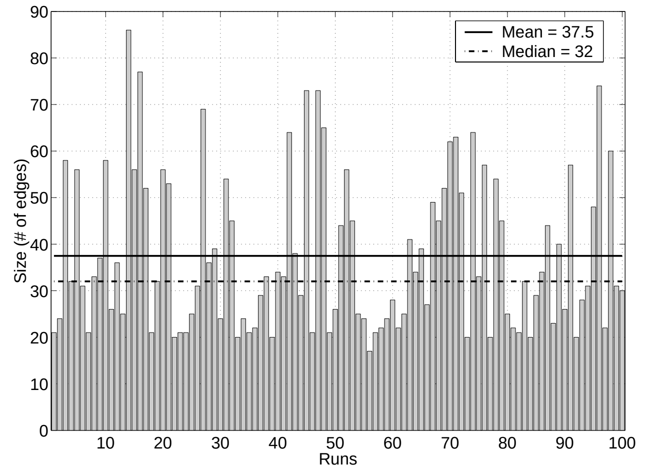

Figure 5: Constant Pars., size of the smallest correct solution.

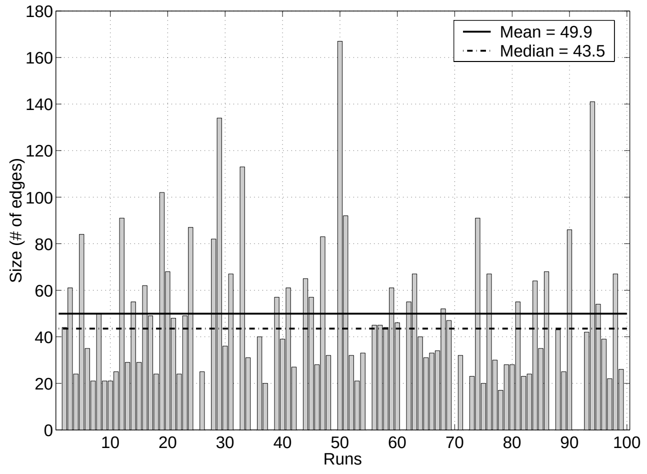

Figure 6: Two Stage, size of the smallest correct solution.

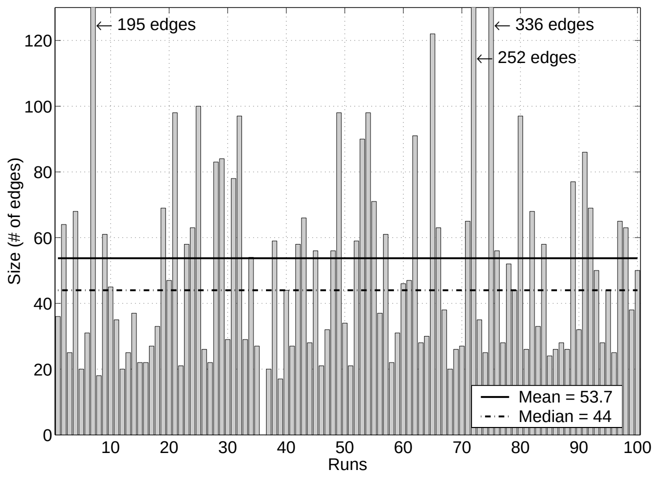

Figure 7: Adaptive Pars., size of the smallest correct solution.

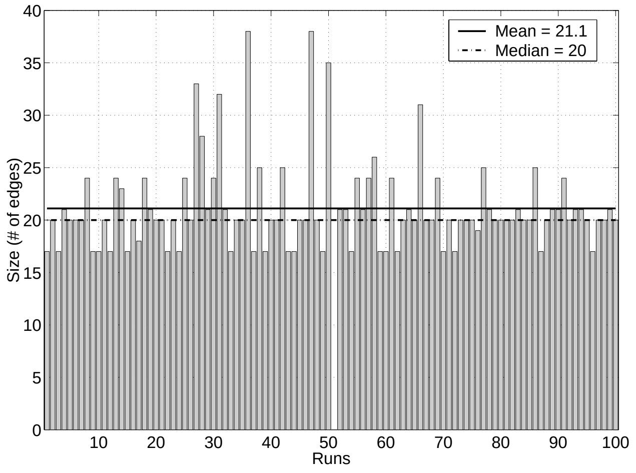

Figure 8: SPEA2, size of the smallest correct solution.

vidual. The two runs for SPEA2 and Constant Parsimony have been started with the same initial population. While SPEA2 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 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 SPEA2 populations at all stages of the evolution good building blocks for correct solutions are present.

6 Conclusions

We have suggested the use of multiobjective optimization for evolving compact GP programs by introducing the program size as a second, independent objective. We have compared a recent multiobjective optimization technique, SPEA2 (an improved version of the Strength Pareto Evolutionary Algorithm), to four other approaches to reduce bloat in GP: Standard GP with tree depth limitation, Constant Parsimony Pressure, Adaptive Parsimony Pressure, and a ranking method (Two Stage) where functionality is optimized first and program size afterwards.

Comparing SPEA2 to the alternative methods we found that:

- It keeps the average tree size lower than any of the other methods.
- It evolves much more compact solutions than all the other methods.
- It is slightly faster in finding solutions than any other of the tested methods.
- Among the other methods Constant Parsimony performs best.
- SPEA2 is well adaptable to problems of different arities without changing any parameters.
- Adaptive Parsimony seems not to be well suited to discrete problems.

We conclude that a Pareto-based multiobjective approach is a promising way of reducing bloat in GP. It is probable that also other Pareto-based multiobjective optimization algorithms would have the observed effects.

Our next steps will focus on investigating this issue on different, discrete and continuous problems further. Moreover, comparisons with other methods like explicitly defined introns (EDI) or automatically defined functions (ADF) would be interesting. Combining multiobjective optimization with these techniques might be another promising direction for future research.

Bibliography

[BFKN98] Wolfgang Banzhaf, Frank D. Francone, Robert E. Keller, and Peter Nordin. *Genetic Programming: An Introduction*. Morgan Kaufmann, San Francisco, CA, 1998.

[Bli96] Tobias Blickle. Evolving compact solutions in genetic programming: A case study. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *PPSN IV*, pages 564–573. Springer-Verlag, 1996.

[BT94] Tobias Blickle and Lothar 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.

[CKO00] D. W. Corne, J. D. Knowles, and M. J. Oates. The pareto envelope-based selection algorithm for multiobjective optimisation. In Marc Schoenauer et al., editor, *PPSN VI*, pages 839–848, Berlin, 2000. Springer.

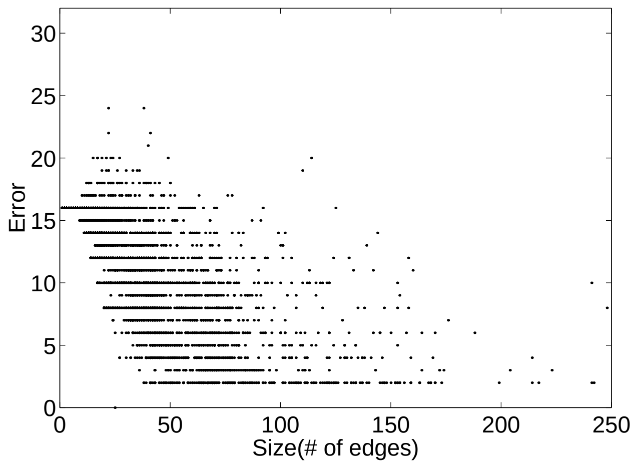

Figure 9: SPEA2 population at generation 30.

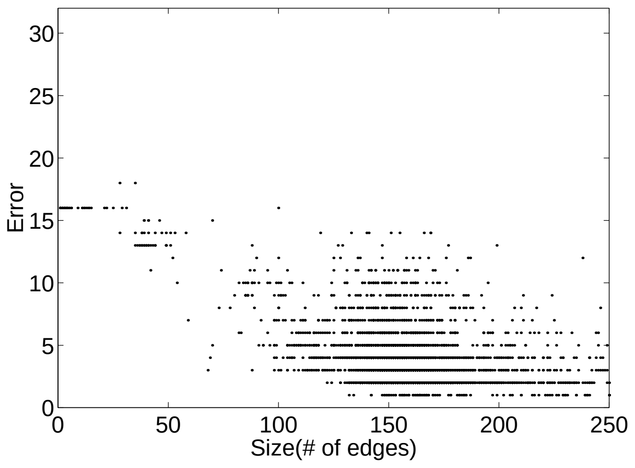

Figure 10: Constant Parsimony population at generation 30.

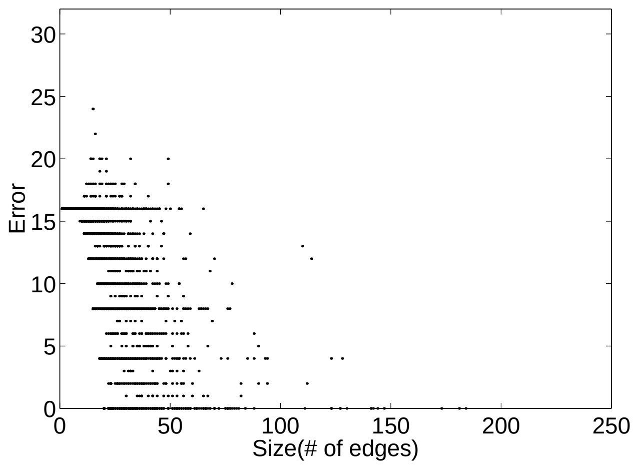

Figure 11: SPEA2 population at generation 200.

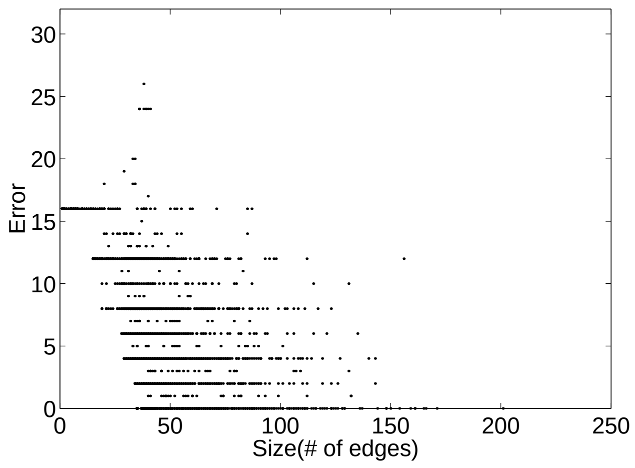

Figure 12: Constant Parsimony population at generation 200.

[KM99] T. Kalganova and J. F. Miller. Evolving more efficient digital circuits by allowing circuit layout evolution and multi-objective fitness. In A.Stoica, D. Keymeulen, and J. Lohn, editors, *Proceedings of the 1st NASA/DoD Workshop on Evolvable Hardware (EH'99)*, pages 54–63, Piscataway, NJ, 1999, 1999. IEEE Computer Society Press.

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, Massachusetts, 1992.

[Koz94] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, Massachusetts, 1994.

[Lan96] William B. Langdon. Data structures and genetic programming. In *Advances in Genetic Programming*, volume 2, chapter 20, pages 395–414. MIT Press, 1996.

[LMBoZ01] Michael Lahanas, Natasa Milickovic, Dimos Baltas, and Nik olaos Zamboglou. Application of multiobjective evolutionary algorithms for dose optimization problems in brachytherapy. In E. Zitzler, K. Deb, L. Thiele, C. A. Coello Coello, and David W. Corne, editors, *Proc. of the first int. conf. on evolutionary multi-criterion optimization (EMO'01)*, volume 1993 of *Lecture Notes in Computer Science*, pages 575–588, Berlin, 2001. Springer-Verlag.

[LP97] W. B. Langdon and R. Poli. Fitness causes bloat. In P. K. Chawdhry, R. Roy, and R. K. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22, London, 1997. Springer-Verlag.

[SF99] Terence Soule and James A. Foster. Effects of code growth and parsimony pressure on populations in genetic programming. *Evoluationary Computation*, 6(4):293–309, 1999.

[ZDT00] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary Computation*, 8(2):173–195, 2000.

[Zit99] Eckart Zitzler. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. PhD thesis, ETH Zurich, Switzerland, 1999. Shaker Verlag, Germany.

[ZLT01] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm. Technical Report 103, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Switzerland, 2001.

[ZM95] Byoung-Tak Zhang and Heinz Mühlenbein. Balancing accuracy and parsimony in genetic programming. *Evoluationary Computation*, 3(1):17–38, 1995.

[ZT99] Eckart Zitzler and Lothar Thiele. Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.