Hierarchical Genetic Programming using Local Modules

Wolfgang Banzhaf
Dept. of Computer Science, University of Dortmund, Germany
Informatik Centrum Dortmund, Germany

Dirk Banscherus
Quantum GmbH, Dortmund, Germany

Peter Dittrich
Dept. of Computer Science, University of Dortmund, Germany

Abstract

This paper presents detailed experimental results for a new modular approach to Genetic Programming, hierarchical GP (hGP) based on the introduction of local modules. A module in a hGP program is context-dependent and should not be expected to improve all programs of a population but rather a very specific subset providing the same context. This new modular approach allows for a natural hierarchy in that local modules themselves may define local sub-modules.

1 Introduction

Genetic Programming is the development of computer programs by evolutionary means [Koz92, BNKF98]. A population of randomly generated programs is subjected to mechanisms of variation and selection in order to arrive at behavior specified by a predefined fitness function. Over the course of the development, programs are generated that more and more approach the desired behavior.

The mechanisms used to vary and select computer programs are similar to those in other areas of evolutionary computation [Fog95], and employ stochastic events as the main driving force for innovation. Mutation and crossover are operators used for variation, proportional or tournament selection are frequently used as selection operators to direct the search process.

As in other fields of evolutionary computation, the representation of the problem is an important aspect of its solution. Genetic Programming originally started with the tree representation of computer programs. Program trees are easy to manipulate by mutation and crossover, and until today they are the most frequently used representation in GP.

Genetic Programming is able to solve an impressive variety of problems from different problem domains [BNKF98]. However, it is well known that there are performance problems with Genetic Programming when tasks grow complex. In such a case, human programmers would rely on a modularization technique allowing them to decompose the task into sub-tasks which are subsequently solved independently, to arrive at a solution by decomposing the solutions of sub-tasks. Some modularization techniques have been proposed for Genetic Programming. Koza has suggested automatically defined functions [4], recently augmented by architecture altering operations [KABK99]. Angeline and Pollack suggest libraries of functions [AP92], Rosca and Ballard adaptive representations [RB94].

It seems, however, that the real break-through for modular Genetic Programming is not yet made.

This paper presents a new modular approach to Genetic Programming (hGP - standing for hierarchical GP) which is based on the introduction of local modules. In contrast to other approaches, our notion of a module in a program is that the context of the module in the calling program is of great importance. A module should not be expected to improve all programs of a population but rather a very specific subset providing the same context. At the same time, our modular approach allows for a natural hierarchy in that local modules themselves may define local sub-modules.

Modules are allowed to evolve at a much slower rate than programs reflecting the need of programs to rely on their modules for improving their function. We discuss this principle which seems to be at work in other natural and artificial modular systems.

Results are presented on a set of discrete and continuous problems, including comparison with regular Genetic Programming.

2 Modular Concepts in Genetic Programming

2.1 The Problem

One of the important issues in Genetic Programming is whether GP is able to scale up. Although there are a number of interesting applications of GP already (see [BNKF98], chapter 12), real world applications suffer from a complexity threshold. It seems that programs of small size may be readily evolvable, but as soon as one gets into hundreds or even thousands of nodes[1], GP becomes less and less effective as a means to generate the targeted function.

A natural method to improve GP performance is therefore the introduction of sub-programs. Partitioning of a problem into sub-problems that can be solved independently is one of the most powerful and general approaches to problem solving that we have developed [AS85]. In Computer Science in particular, where problems of large complexity are solved daily, modularization is a key enabling technology for progress. Many of the biggest steps in software and hardware development over the last decades may be traced back to the introduction of modularization / hierarchization techniques.

Thus, one of the big challenges for genetic programming may be formulated as this: Is it possible for a Genetic Programming system to evolve modular solutions to problems *automatically*? Note the emphasis on "automatically". It is clear that a manual specification of sub-problems will work, provided the sub-problem complexity is sufficiently small to be treated by regular GP. However, will it be possible to delegate the structuring of the problem to an automatic process like GP?

2.2 Existing Approaches

Mainly three approaches have been proposed in the course of the last decade to solve the problem of modularization by Genetic Programming, "automatically defined functions"

[1]Three nodes in a tree usually correspond to one line of code.

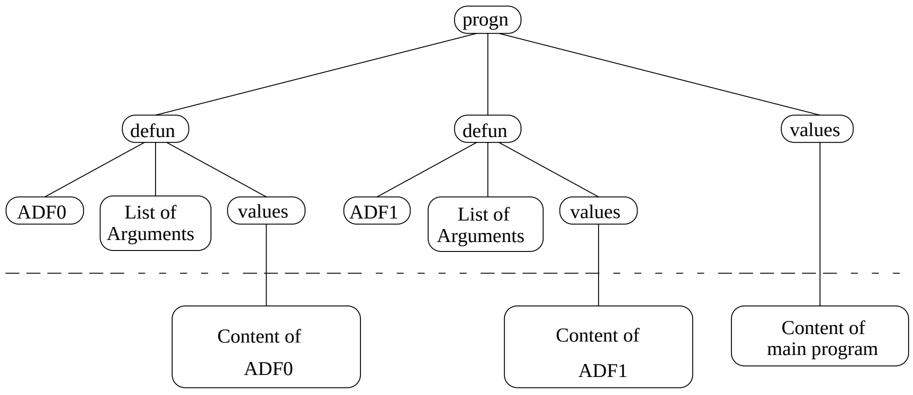

Figure 1: Structure of program using 2 automatically defined functions, ADF0 and ADF1

(ADFs) [Koz94], evolutionary module acquisition [AP92] and adaptive representation [RB94]. This section will briefly summarize these approaches.

2.2.1 Automatically defined functions

Automatically defined functions are the most widely used method of modularization in genetic programming to date. ADFs typically are predetermined in a couple of aspects before a run can start using them: Name of ADF, Number of arguments of an ADF, Set of functions for an ADF, and Set of terminals for an ADF.

Simply put, ADFs have a form and a function. The function is evolvable the form is not. Technically, ADFs consist of two different parts, a function definition part and and a function evaluation or work performing part. Evolution during a run only takes place in the work performing part. Figure 1 gives an idea of the principle. In [Koz94] Koza shows that ADFs are advantageous in many problems of a complexity increased in comparison to standard GP problems, e.g. the 6-parity problem. A comparison is based on the computational effort (in the number of fitness evaluations) necessary to solve an instance of such a problem with 99 % probability. Complexity of a tree is measured by counting the nodes in that tree, with assigning one node only to each of the ADFs.

The fixed structure of ADFs is a mixed blessing. On the one hand, it *requires* the user of the GP system to specify before-hand the number and features of ADFs, on the other hand, it *allows* the user to identify important elements of an anticipated successful solution to the problem that the evolutionary process should make use of in its search. Addressing the downside, Koza has recently proposed architecture altering-operations, that manipulate the function-defining structure of an ADF as well as the number of ADFs allowed [KABK99].

2.2.2 Module Acquisition

Module acquisition is another method to modularize genetic programming. Here, two additional operators are added to the system that allow to manipulate trees by compressing and expanding nodes. Compression of nodes takes place when a subtree is isolated and substituted by a node with a unique name. Everything below a prespecified depth of the subtree is considered argument to that new node. As a result, the function set of the problem is enlarged by a newly defined function. This function is stored in a genetic

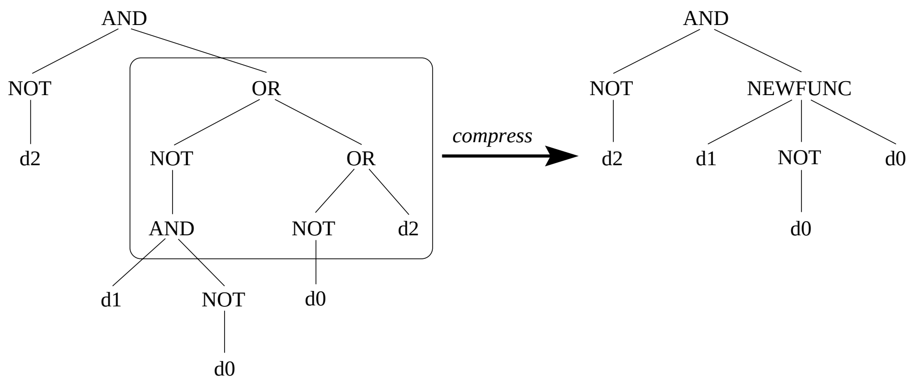

Figure 2: The compression in module acquisition. NEWFUNC is added to the genetic library for further use by the population.

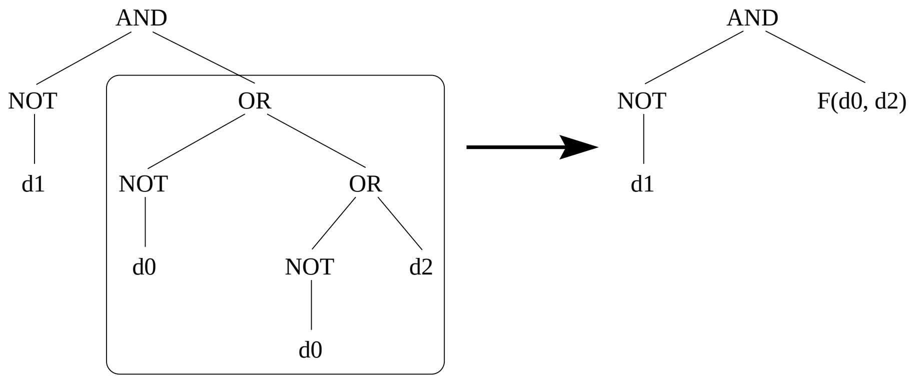

Figure 3: Adaptive representation. F is considered a module with the same arguments as the full tree.

library for use by other population members as well. A sketch of the procedure is shown in Figure 2. Nodes can be used repeatedly, thus a hierarchic structure of modules is possible.

The opposite action is taken by the expansion operator which takes a node from the genetic library and expands it again. Angeline and Pollack who introduced the method offer no striking conclusion as to whether their approach is advantageous. Kinnear states that it is not saving space or time [Kin94], but his study is based on one problem only. It remains to be seen whether the idea of module acquisition can be used efficiently.

2.2.3 Adaptive Representation

In the approach by Rosca and Ballard [RB94], full subtrees only are allowed to be used as modules. Figure 3 gives an idea of the procedure.

Modules in AR are selected according to criteria that are tied to the performance of the individual or parts of it. Rosca and Ballard discuss some variants of performance measures and show that a considerable improvement in evolution speed is possible with their approach. In addition, however, to the introduction of a modular concept, AR works with epochs of evolution, where at the beginning of each epoch a number of individuals in the population is substituted by newly generated individuals that make use of the modules generated in the last epoch. It thus remains unclear where the advantage of AR comes from.

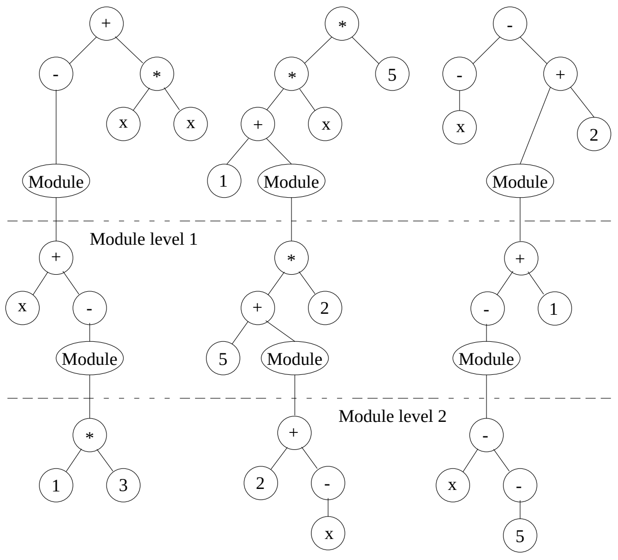

Figure 4: Example of three hierarchical levels of evolution in hGP. Modules on each level evolve in their own level and are called from the next higher level.

2.3 Local, context-sensitive modules: hGP

We shall introduce here another general method for specifying modules. The idea of local and context-sensitive modules is motivated by the success of Gruau's work on cellular encoding [Gru93]. At the surface, cellular encoding is about making graphs available for use with genetic programming. Gruau develops neural networks, other researchers develop other graph-like applications, e.g. electric circuits [KBA+97].

The aspect interesting here, however, is that of hierarchical evolution. We use a number of hierarchical levels of evolution, with a population on each of them. On the highest level, individuals of the population evolve their functionality. On the lower levels, modules of level 1 ... n evolve through the same mechanisms of variation and selection. Figure 4 depicts the situation. Modules on higher levels (including the individuals on the highest level) are able to call modules of the next lower level as subprograms.

As in ADFs, the newly defined modules are local to an individual. They are not available to the population as a whole but only to the one individual which has called them. Thus, an individual has to evolve a good choice of modules completely for itself, only taking help through crossover of material from other individuals having defined modules at the same level. Much as an entire GP system has global convergence to a solution, so do the local modules have a tendency to converge, even without being able to be accessed by all individuals.

Arbitrary crossover of material is forbidden in this method. Rather, modules at the same level of description are able to exchange material. Koza has called this method structure preserving crossover [Koz92]. ADFs make use of this method, too, since the two types of branches in an ADF are only allowed to be crossed over with their kin.

What is the difference, then, between modules on different levels? An important difference lies in the fact that modules on different levels evolve with different speed. Similar to the compress operation in module acquisition, which explicitly forbids further evolution of material that has been compressed, speed of evolution is the key difference. The radical step of freezing the compressed material completely is substituted, however, by a less radical, but more general step: to decrease the speed of evolution. The lower in the hierarchy a module is located, the slower it is allowed to evolve. Although this is somewhat counter-intuitive at first glance, it is indeed the method which Nature used when evolving modules. The more fundamental the modules are the less evolution Nature allows at that level. The appearance of the genetic code is a typical example of this phenomenon [Osa95], the development of repair mechanisms in the replication of genetic material is another [FWS95].

Thus, our method to evolve modules at different levels will be to adjust the speed of evolution. The lower in the hierarchy, the less crossover and mutation events will hit them. In a nutshell, higher level modules can be discerned from lower level modules by their larger speed of evolution. Interestingly if we turn this argument around, another observation in Nature seems to fit in very well with this picture: Higher level modules, i.e. modules commanding higher complexity must be faster in evolution if there is no way to reduce evolution speed in lower levels, i.e. to stabilize developments there.

The generation of a lower level module in hGP is done during the evolution of the higher level individual: After crossover, modules are identified in the best individuals of a population only. Modules are formed by search for valuable sub-trees in these individuals. The general method for finding valuable subtrees is to compute the differential fitness [RB96, Ros95b] with and without the subtree under discussion (Sec. 3). Ranking selection is then applied to identify the best subtrees and generate a module of them in the next lower level. Various parameters determine this procedure, like e.g. maximum number of modules per individual, maximal depth for computation of differential fitness, etc.

Since on the lower level evolution should progress, too, a fitness must be assigned to each of the newly created modules. In hGP, the fitness of a module is exactly the same as the fitness of the individual which is calling it in the next higher level. Thus, a good program will automatically transfer its high fitness to the module used by it.

Crossover and mutation on lower module levels work similar as on higher levels. hGP also allows different variants, e.g. based on homology and quality of subtrees. hGP was implemented as an extension of gpc++0.4.

2.3.1 The hGP Algorithm

In pseudo code the algorithm executed for each generation in hGP reads:

```
FOR level := 0 TO maxLevel DO
  DO popSize(pop[level]) * evolutionSpeed[level]TIMES
    (mum, dad, child) := Selection(pop[level])
    Crossover(mum, dad, child)
    Mutate(child, mutationStrength[level])
    IF level < maxLevel - 1
      ModuleList := searchModules(child)
      AddModules(pop[level], ModuleList)
    FI
  OD
OD
```

3 Identification of Valuable Modules

One important problem in hGP is how to find good modules. This function is implemented by `searchModules(...)` above. Our general approach is to measure the value of a system component by exchanging the component by a neutral component [Ros95b]. The following variants differ in the way a neutral component is generated:

- *Constant value.* The subtree is replaced by a global constant value.
- *Intron.* The subtree is replaced by a randomly generated intron.
- *Random constant.* The subtree is replaced by a random constant. A new random constant is drawn for every subtree that has to be rated.
- *Random stream* (many random values). The individual is evaluated many times, while the subtree is replaced by different randomly drawn constants.

Figure 5 shows a comparison of the module rating techniques *constant value*, *random constant*, and *intron*. *Random stream* was discarded after preliminary experiments did not show its effectiveness. From the figure it can be observed that *constant value* has the worst performance. The methods *random constant* and *intron* showed comparable performance. Although Fig. 5 indicates a better performance for the *intron* method in the continuous case (regression) and a better performance for the *random constant* method in the discrete case (even-7-parity), a general conclusion concerning which method is preferable should not be drawn based on only two problem instances. For the experiments in Sec. 5 the method showing the best performance for the respective case is applied.

4 Performance Measure

To calculate the run time performance of an algorithm one has to assign a duration time to every statement (operation) of the algorithm. An easy and typical approach is to identify the most time consuming operations and to assign a constant value of 1 to them. All other operations are considered to have a duration time of 0. In a time complexity analysis of sorting algorithms, for instance, one assumes that each comparison operation takes 1 unit of time and all other operations 0.

In most evolutionary algorithms time is measured in terms of the number of fitness evaluations. This model of computation time assumes that every fitness evaluation requires constant time (e.g. 1 time unit) and other operations require 0 time. These assumptions are reasonable in most conventional GAs or ES with fixed length representations. In GP, however, as well as in other length-changing EAs, these assumptions are not adequate and may lead to wrong conclusions.

As an example consider the following regression experiments with and without ADFs: The runs with ADFs show a faster convergence in terms of fitness evaluation, e.g. 464,000 evaluations compared to 6,528,000 fitness evaluations in runs without ADFs. However, the evaluation time of an individual with ADFs is 21 times longer (in terms of node evaluations) than an individual without ADFs. In this case the conclusion based on the time measurement in terms of fitness evaluations is wrong.

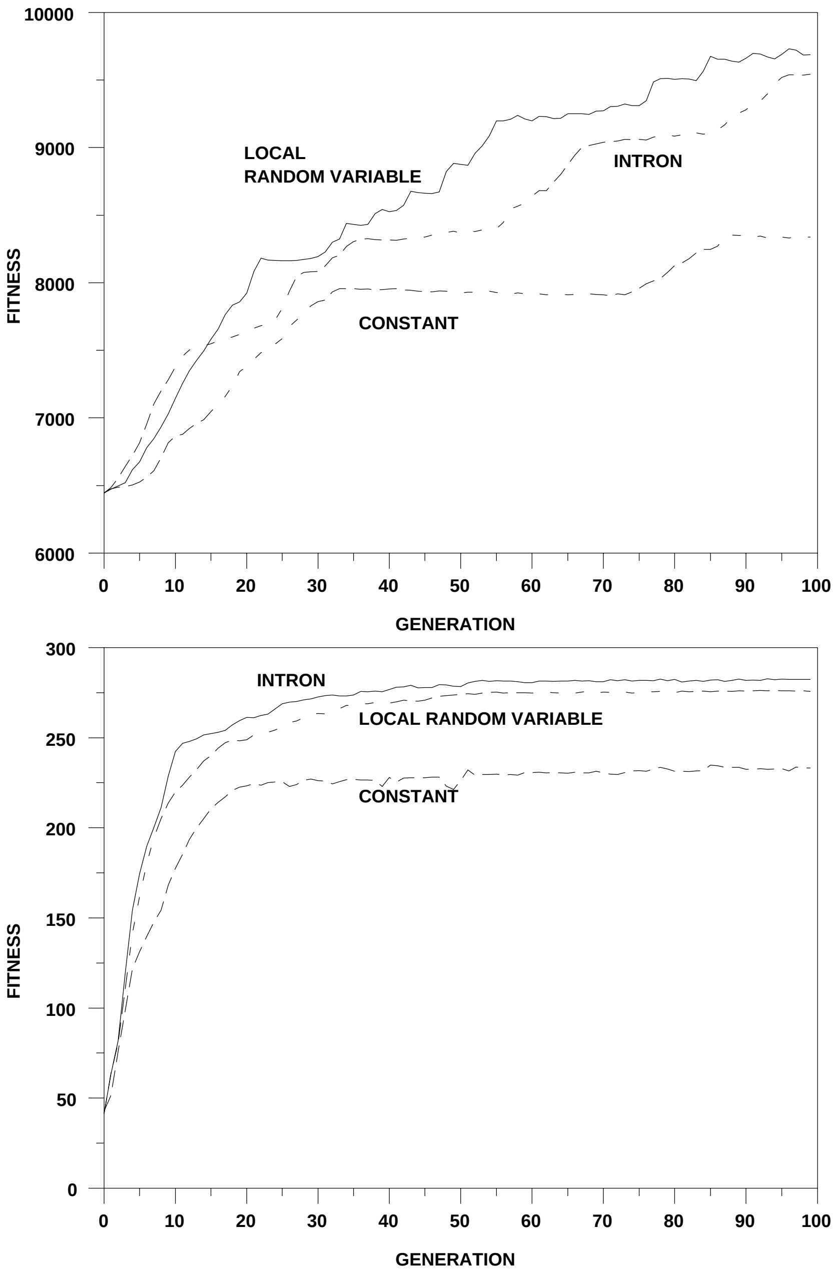

Figure 5: Average fitness for different module rating strategies. Upper: Even-7-parity, Lower: Regression problem on f_4. 30 runs for each strategy and problem.

In order to circumvent these problems we use the number of evaluated nodes as a measurement for execution time. This is reasonable, because in our case the evaluation of every operator and constant takes about the same time. In general, it might become necessary to distinguish the evaluation time of different operators.

It should be noted that there are also GP scenarios where simpler models of time and others where more complex models of time have to be applied, e.g. in case of online evolution [DBB98] or sub-tree sharing. Therefore, we think that it is always wise to question the time model before drawing any conclusion related to the speed of a GP system. In our opinion a time model is reasonable for drawing certain conclusions if and only if these conclusions are invariant when the time model is exchanged by a model with higher precision.

5 Results with Hierarchical Genetic Programming (hGP)

For the following experiments hGP has been substantially restricted. Two variants referred to as *hGPminor* and *hGP* are tested with the following restrictions:

- Number of modular levels: Only level 1 modules allowed
- Number of modules per calling individual: Only 1 module allowed
- Mutation only on highest level allowed

In *hGPminor* evolution on module level is not allowed. In this case the generation of modules works mainly as a protection of valuable code against mutation and crossover.

In *hGP* evolution on the module level is allowed. The settings for the evolution on the module level are:

- Crossover variant: replace a bad subtree by a randomly selected subtree
- Crossover probability on module level: 33%. Thus, `evolutionSpeed[0] = 1.0; evolutionSpe`

Surprisingly, the crossover variant “replace a bad subtree by a good subtree” has led to significantly worse results. Experiments have also confirmed that hGP is robust concerning the setting of the crossover probability on module level (up to 50 %).

5.1 Test Problems

We report on six test problems here that have been used (Tab. 1), to compare the performance of hGP with standard GP: 4 continuous problems from function regression (Fig. 6 and 7) and two instances of the discrete even-N-parity problem [Koz92] with $N = 5$ and $N = 7$.

Even-5-parity, even-7-parity and regression on f_4 have been used during the development process of hGP and extensive experiments have been carried out based on these problems [Ban98]. Regression problems on f_1, f_2, and f_3 are used after development of hGP for validation.

Problem	Type	Symbol	Regression function
1	continuous	f_1	randomly selected y-values (Fig. 6, left)
2	continuous	f_2	steps (Fig. 6, right)
3	continuous	f_3	$x^6 - 4x^5 - 3x^4 + 4x^3 - 2x^2 - x + 4$
4	continuous	f_4	$\frac{x^3-x^2-x+3}{x+\frac{5}{9}}$
5	discrete		even-5-parity
6	discrete		even-7-parity

Table 1: Test problems used here to measure the performance of hGP.

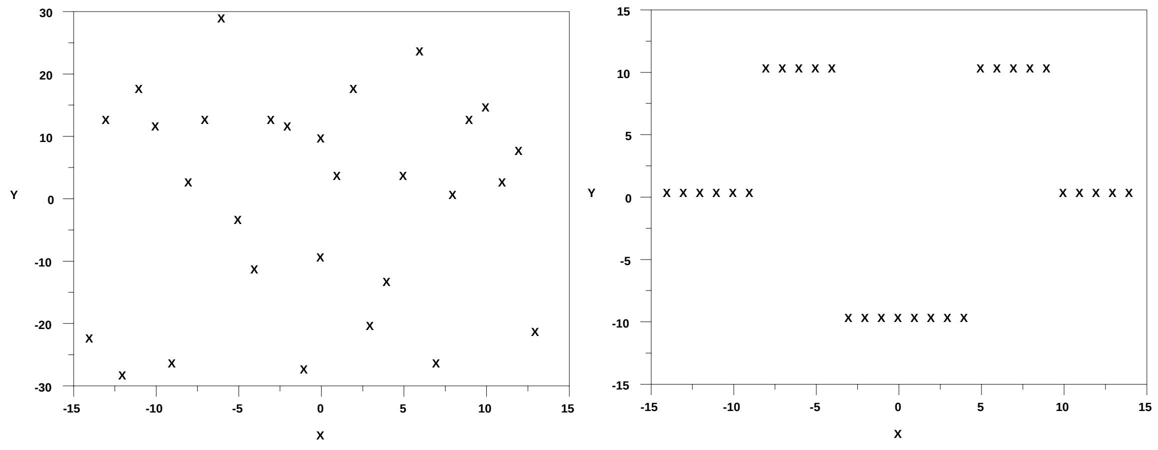

Figure 6: Test functions f_1 (left) and f_2 (right).

Tables 2 and 3 give the run parameters of these test runs in overview.

We compared standard GP, ie. Genetic Programming without modules, and hGP, without (hGPminor) and with (hGP) evolution on the module level. In preliminary experiments a reduction of evolution speed to about 1/3 that at the level of individuals turned out to be efficient, although different applications shall require different module evolution speed. In another application, we were successful with a speed of 1/10 that at the higher level programs [OBN96]

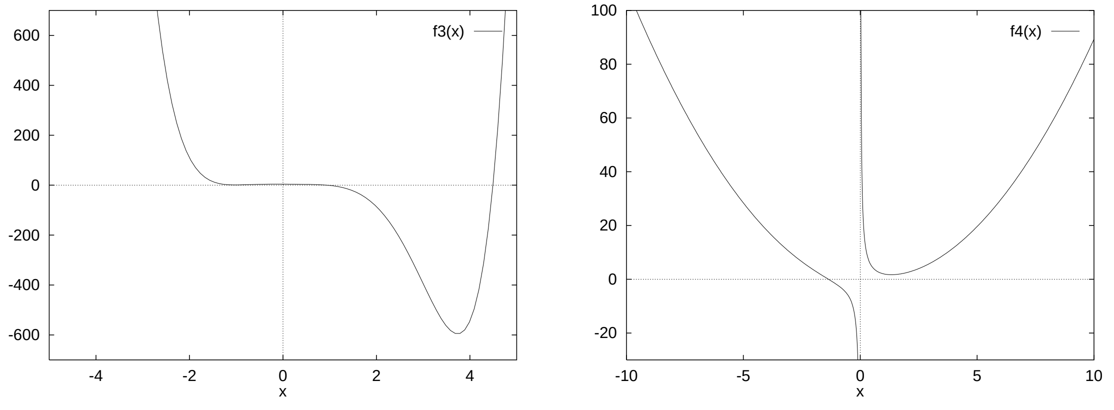

Figure 7: Test functions f_3 (left) and f_4 (right).

Parameter	Setting
population size	3 000
selection	(10,1)-tournament
generation equivalents	100
crossover-frequency on top level	100 %
crossover-frequency on module level	0 % (hGPminor) 33 % (hGP)
mutation-frequency on top level maximum tree depth	2 % 17
maximum initial tree depth	6
initialization	ramping half and half
maximum number of modules per individual	1
problem	function regression for f
raw fitness	$\Phi = 100 * \sum_i max(10, \|\|f(x_i) - y_i\|\|$
parsimony term	$100 \cdot (1 - \frac{10}{10+\kappa(a_i)}$
terminal set	$T = \{0, 1, ...9, x\}$
function set	$F = \{+, -, *, /_0\}$
termination-criterion	exceeding the maximum number of generations

Table 2: The Koza tableau of parameter settings for the regression problem in hGP. Comparison with standard GP containing no modules. $\kappa(a_i)$ is the expanded structural complexity of the individual a_i [Ros95a].

Parameter	Setting
population size	3 000
selection	(10,1)-tournament
generation equivalents	100
crossover-frequency on top level	100 %
crossover-frequency on module level	0 % (hGPminor) 33 % (hGP)
mutation-frequency on top level maximum tree depth	2 % 17
maximum initial tree depth	6
initialization	ramping half and half
maximum number of modules per individual	1
problem	even-N-parity, N = 7
raw fitness	$\phi = 100 * (numberofmismatches)$
parsimony term	$100 \cdot (1 - \frac{10}{10+\kappa(a_i)}$
terminal set	$T = \{D_0, D_1, ..., D_N\}$
function set	$F = \{AND, OR, NAND, NOR\}$
termination-criterion	exceeding the maximum number of generations

Table 3: The Koza tableau of parameter settings for the even-N-parity problem in hGP. Comparison with standard GP containing no modules. $\kappa(a_i)$ defined as above.

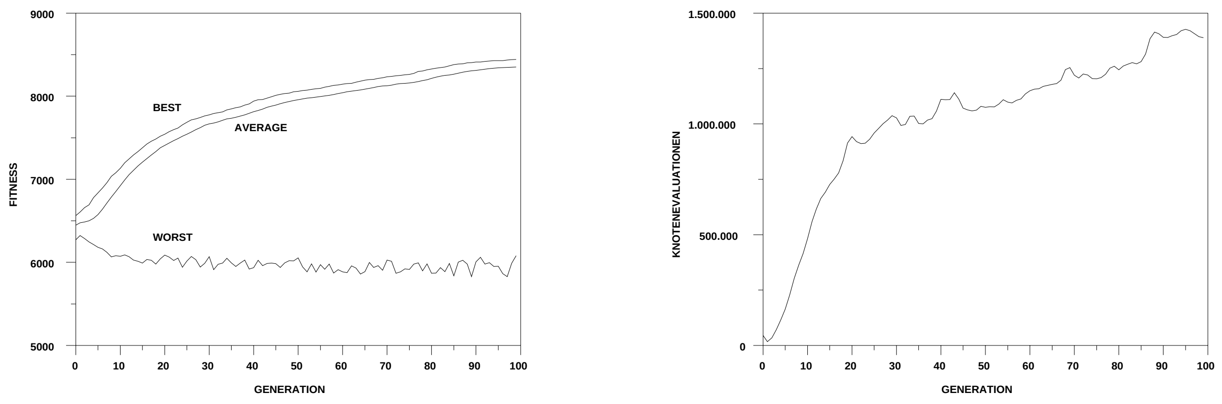

Figure 8: Even-7-parity, standard GP. 50 runs. Left: best, average, and worst fitness over time (measured in generation). Right: node evaluations per generation.

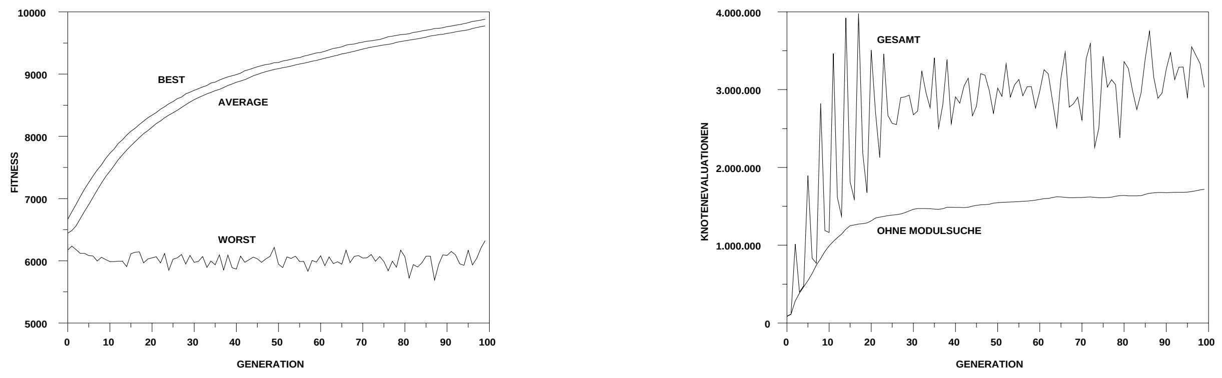

Figure 9: Even-7-parity hGP minor. 50 runs. Left: best, average, and worst fitness over time (measured in generation). Right: node evaluations per generation. Lower curve shows the node evaluation needed only for fitness evaluation. The upper curve shows the node evaluation needed for fitness evaluation and module search. The area between the upper and the lower curve represents the additional effort which is spended for searching good modules.

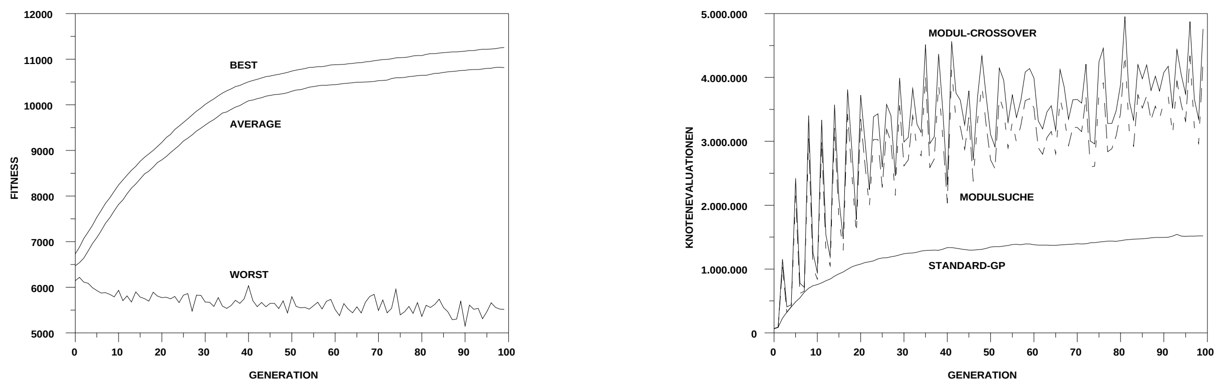

Figure 10: Even-7-parity hGP. 50 runs.

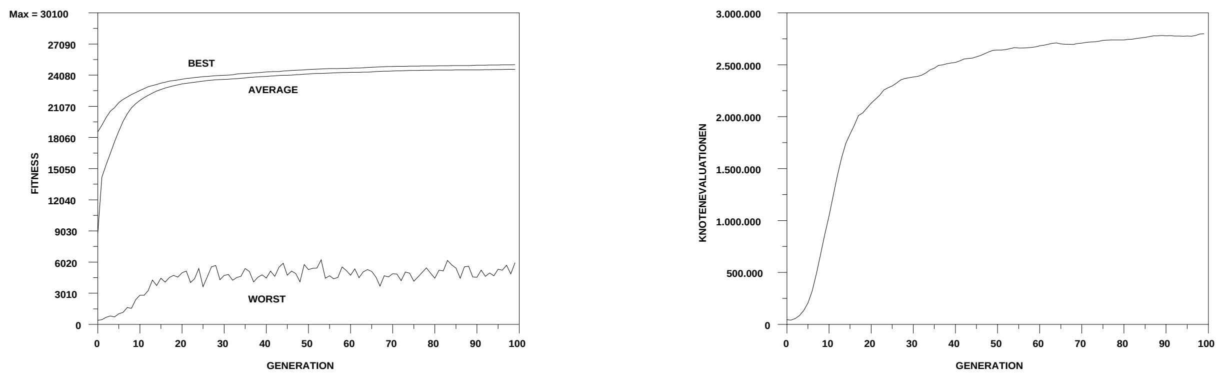

Figure 11: Regression on f_4, standard GP. 50 runs. Left: best, average, and worst fitness over time (measured in generation). Right: node evaluations per generation.

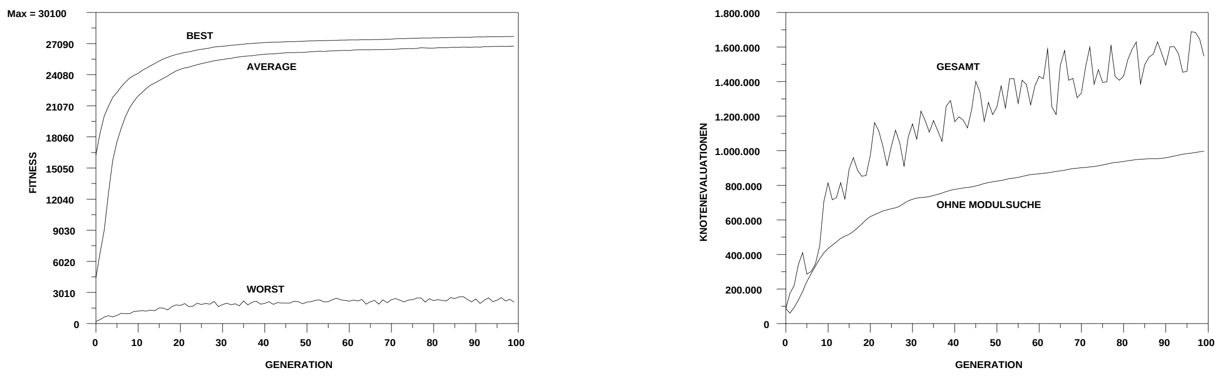

Figure 12: Regression on f_4, hGP minor. 50 runs. Left: best, average, and worst fitness over time (measured in generation). Right: node evaluations per generation. Lower curve shows the node evaluation needed only for fitness evaluation. The upper curve shows the node evaluation needed for fitness evaluation *and* module search.

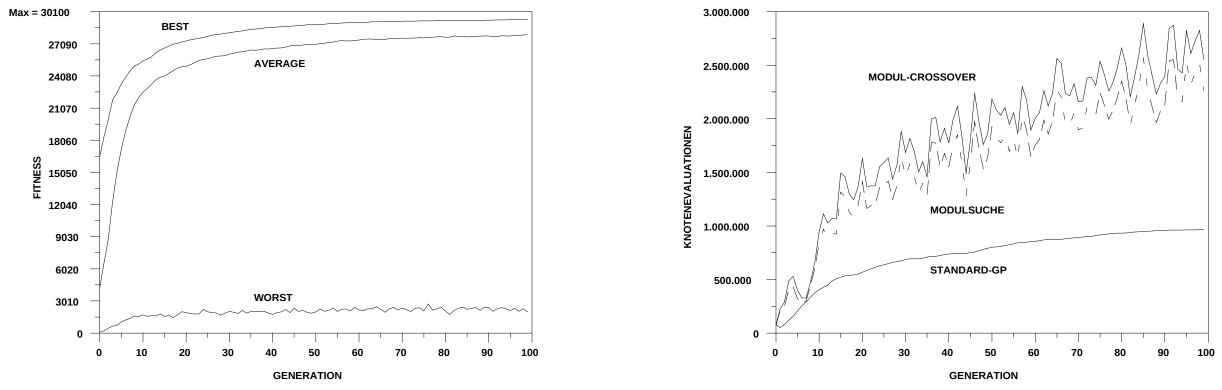

Figure 13: Regression on f_4, hGP. 50 runs.

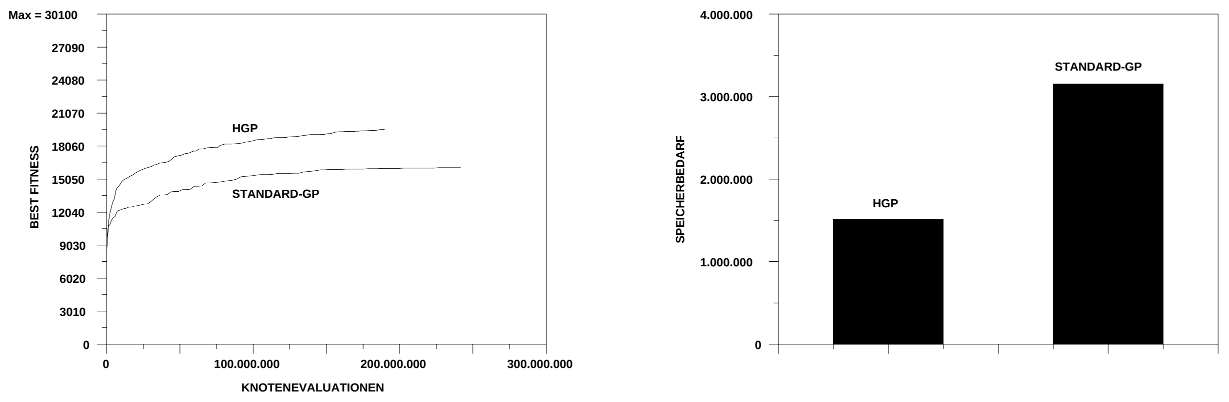

Figure 14: Regression on f_1, standard GP vs hGP. 30 runs each. Left: Best fitness over time. Time is measured in node evaluations. Right: memory consumption.

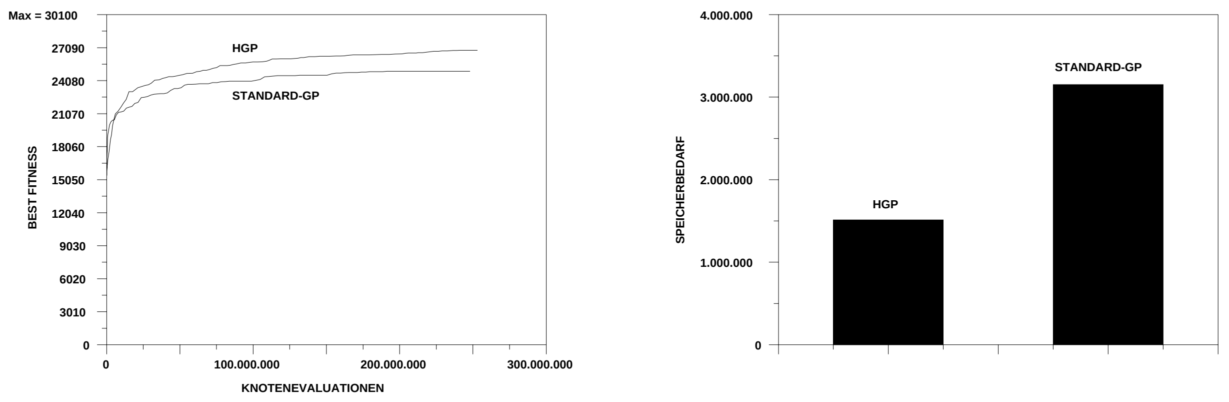

Figure 15: Regression on f_2, GP vs hGP. 30 runs each.

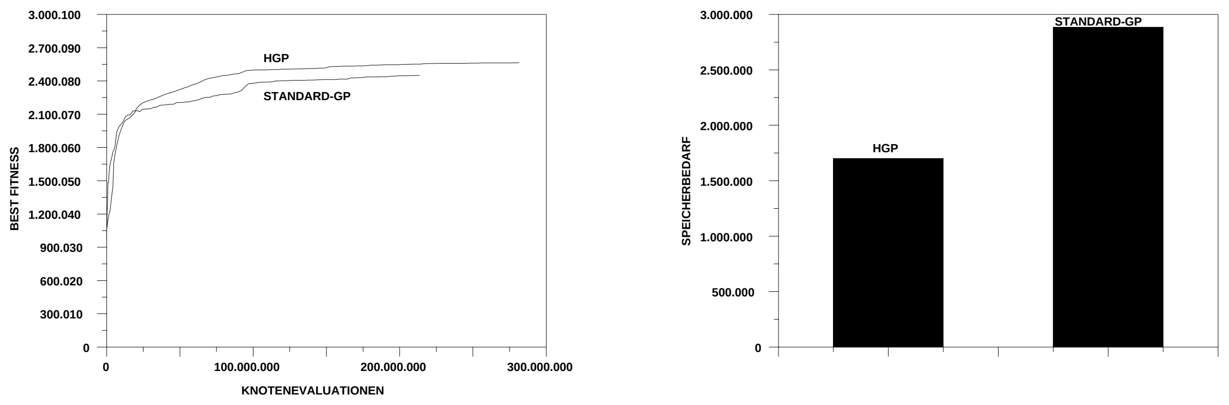

Figure 16: Regression on f_3, GP vs hGP. 30 runs each.

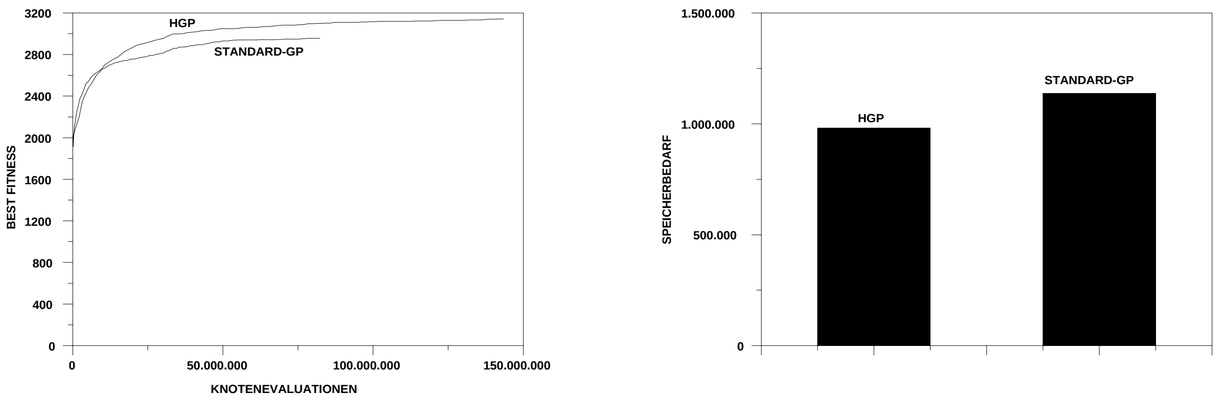

Figure 17: Even-5-Parity, GP vs hGP. 30 runs each. Left: Best fitness over time. Time is measured in node evaluations. Right: memory consumption.

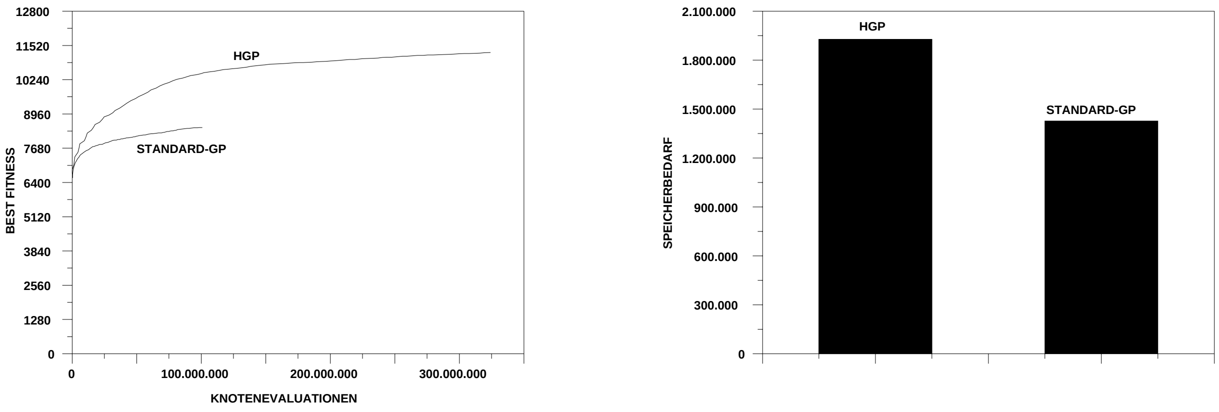

Figure 18: Even-7-Parity, GP vs hGP. 30 runs each.

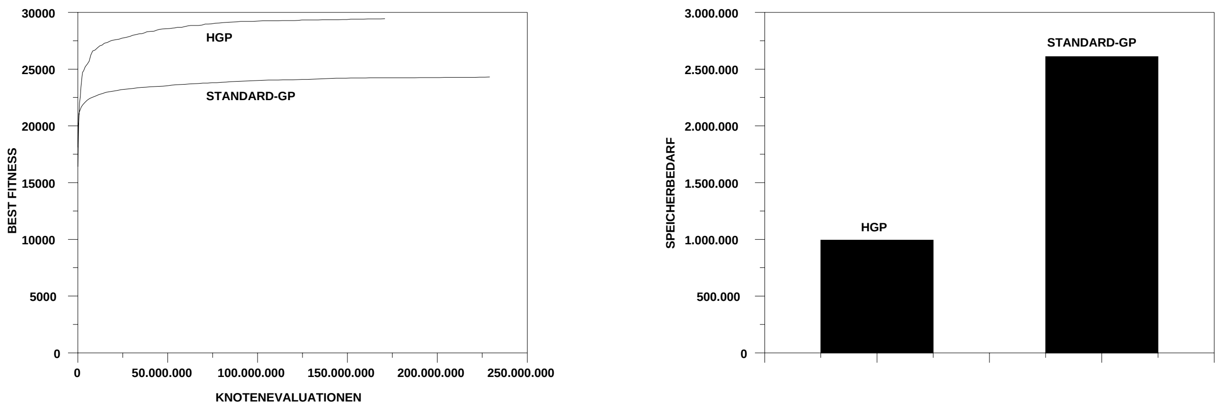

Figure 19: Regression on f, GP vs hGP. 30 runs each.

6 Discussion

Figures 8-13 show the detailed performance of standard GP, hGPminor and hGP for the test problems 4 (regression on f_4) and problem 7 (even-7-parity). In addition to best, average, and worst fitness the nodes evaluated per generation is depicted in the right figures. It can be seen that in both cases hGPminor outperforms standard GP and hGP outperforms hGPminor. The nodes evaluated per generation increase in all cases which reflects that average individual length is growing. The growing process is bounded because a parsimony pressure is activated. Note, that the parsimony pressure is very weak. E.g. for the parity problem it has only an effect, if two individuals represent exactly the same function.

For a fair comparison of convergence speed in Fig. 14-19 time is now measured in node evaluations. For all 6 test problems hGP outperforms standard GP. The performance gain depends on the problem. In some cases its obvious in some cases only marginal. Figures 14-19 compare also the memory consumption of standard GP vs. hGP. In general hGP does not consume significantly more memory than standard GP. In many cases its memory consumption is even smaller.

The performance gain in hGPminor is achieved because good modules are found. This has been shown (not here) by a neutral model where module are generated by randomly selecting a subtree which shows a worser performance. Why does hGPminor reaches better fitness values than standard GP ? The module generation implies a proliferation of "good" and locally valuable code. This code seems to be also globally valuable.

7 Conclusion

hGP shows good performance even when a more detailed time model – number of node evaluation – is applied. The performance gain is based on efficient module search techniques which are based on the differential fitness calculated by replacing the designated module by a neutral structure. Whether a larger number of levels increases the performance of hGP is still an open question and should be a subjects for future investigations.

ACKNOWLEDGMENT

Support has been provided by the Deutsche Forschungsgemeinschaft within the Sonderforschungsbereich 531, project B2.

References

[AP92] Peter J. Angeline and J. B. Pollack. The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*. Lawrence Erlbaum, 1992.

[AS85] Harold Abelson and Gerald Sussmann. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, 1985.

[Ban98] Dirk Banscherus. *Hierarchische Genetische Programmierung mit lokalen Modulen.* diploma thesis, Dept. of Computer Science, University of Dortmund, Informatik XI, D-44221 Dortmund, Germany, 1998.

[BNKF98] Wolfgang Banzhaf, Peter Nordin, Robert Keller, and Frank D. Francone. *Genetic Programming — An Introduction.* dpunkt/Morgan Kaufmann, Heidelberg/San Francisco, 1998.

[DBB98] Peter Dittrich, Andreas Buergel, and Wolfgang Banzhaf. Learning to move a robot with random morphology. In Phil Husbands and Jean-Arcady Meyer, editors, *Evolutionary Robotics, First European Workshop, EvoRob98*, pages 165–178. Springer, Berlin, 1998.

[Fog95] David Fogel. *Evolutionary Computation.* IEEE Press, Piscataway, NY, 1995.

[FWS95] Errol C. Friedberg, Graham C. Walker, and Wolfram Siede. *DNA Repair and Mutagenesis.* ASM Press, New York, 1995.

[Gru93] Frederic Gruau. Genetic synthesis of modular neural networks. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 318–325, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[KABK99] John R. Koza, David Andre, Forrest Bennett, and Andrew Keane. *Genetic Programming III.* Morgan Kaufmann, San Francisco, CA, 1999.

[KBA+97] John R. Koza, Forrest H Bennett III, David Andre, Martin A. Keane, and Frank Dunlap. Automated synthesis of analog electrical circuits by means of genetic programming. *IEEE Transactions on Evolutionary Computation*, 1(2):109–128, July 1997.

[Kin94] Kenneth Kineer. Alternatives in automatic function definition: A comparison of performance. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, pages 119–141. MIT Press, Cambridge, MA., 1994.

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

[Koz94] John R. Koza. *Genetic Programming II.* MIT Press, Cambridge, MA, 1994.

[OBN96] Markus Olmer, Wolfgang Banzhaf, and Peter Nordin. Evolving real-time behavior modules for a real robot with genetic programming. In *Proceedings of the international symposium on robotics and manufacturing*, Montpellier, France, May 1996.

[Osa95] Suyozo Osawa. *Evolution of the Genetic Code.* Oxford University Press, Oxford, 1995.

[RB94] Justinian P. Rosca and Dana H. Ballard. Learning by adapting representations in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Orlando, Florida, USA*, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[RB96] Justinian P. Rosca and Dana H. Ballard. Evolution-based discovery of hierarchical behaviors. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*. AAAI / The MIT Press, 1996.

[Ros95a] Justinian P. Rosca. An analysis of hierarchical genetic programming. Technical Report 566, University of Rochester, Rochester, NY, USA, 1995.

[Ros95b] Robert Rosen. *Life Itself.* Columbia UNiversity Press, New York, 1995.