Chapter 1

Hierarchical Genetic Programming using Local Modules

Wolfgang Banzhaf
Dept. of Computer Science, University of Dortmund, Germany
Dirk Banscherus
Quantum GmbH, Dortmund, Germany
and Peter Dittrich
Dept. of Computer Science, University of Dortmund, Germany

This paper presents 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 recursiveness in that local modules themselves may define local sub-modules.

1.1 Introduction

Genetic Programming is the development of computer programs by evolutionary means [9, 5]. A population of randomly generated programs is subjected to mechanisms of variation and selection in order to arrive at behavior specified by an explicit or implicit 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 [6], 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 [5]. 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 recomposing 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 [11]. Angeline and Pollack suggest libraries of functions [2], Rosca and Ballard adaptive representations [17]. 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. More details can be found in [4].

1.2 Modular Concepts in Genetic Programming

1.2.1 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 [5], 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 which can be solved independently is one of the most powerful and general approaches to problem solving that we have developed [1]. 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?

1.2.2 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“ (ADFs) [10], evolutionary module acquisition [2] and adaptive representation [17]. A more detailed discussion can be found in [5, 4]

1.2.3 Local, context-sensitive modules: hGP

We introduce 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 [8]. 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 [12].

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 1.1 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. Each level has its own set of terminals and functions.

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

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

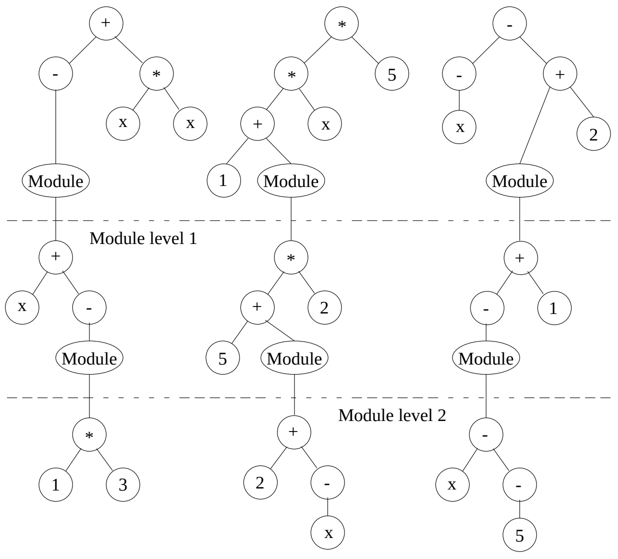


Figure 1.1: Example of three hierarchical levels of evolution in hGP. Modules on each level evolve in their own level and are called from the next higher level.

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 [9]. 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.

Apart from the potential difference in terminal and function sets, 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 [14], the development of repair mechanisms in the replication of genetic material is another [7].

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 [16, 18] with and without the subtree under discussion. Ranking selection is then applied to identify the best subtrees and to generate a module for 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.

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
```

1.3 First 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 agains 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% (i.e., evolution speed on module level is 1/3).

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 %).

We studied six test problems that have been used to compare the performance of hGP with standard GP: 4 continuous problems from function regression and two instances of the discrete even-N-parity problem [9] with $N = 5$ and $N = 7$. The time measurement performance was based on the number of node evaluations, not on the number of individuals or generations evaluated (see [4]).

Problem	Type	Symbol	Regression function
1	continuous	f_1	randomly selected y-values
2	continuous	f_2	steps
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

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 [3]. Regression problems on f_1, f_2, and f_3 are used after development of hGP for validation. Table 1.1 gives the run parameters. Due to a lack of space we report here only on the even-7-parity problem. The reader may compare [4] for the performance on other problems.

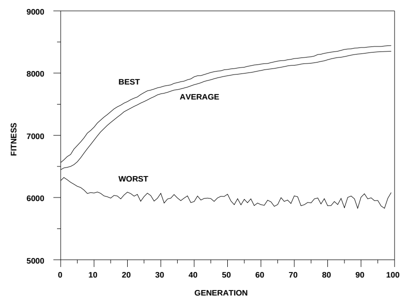
We compared standard GP, i.e. Genetic Programming without modules, and hGP, without (hGPminor) and with evolution on the module level. In preliminary experiments a reduction of evolution speed to about 1/3 that at the level of individuals turned out to be efficient, although different applications shall require different module evolution speed. In another application, we were successful with a speed of 1/10 of that of the higher level programs [13]

1.4 Discussion and Conclusion

Figures 1.2-1.5 show the performance of standard GP, hGPminor and hGP for the test problems 6 (even-7 parity). In addition to best, average, and worst fit-

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	$\phi = 100*$ (number of mismatches)
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 1.1: The Koza tableau of parameter settings for the even-N-parity problem in hGP. Comparison with standard GP containing no modules. $\kappa(a_i)$ is the expanded structural complexity of the individual a_i [15].



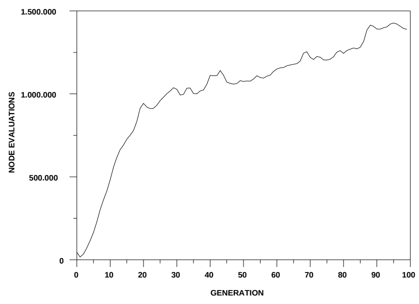


Figure 1.2: Even-7-parity Standard GP. Left: Best, average and worst fitness over time. Right: Number of node evaluations accumulated over time. Average of 50 runs.

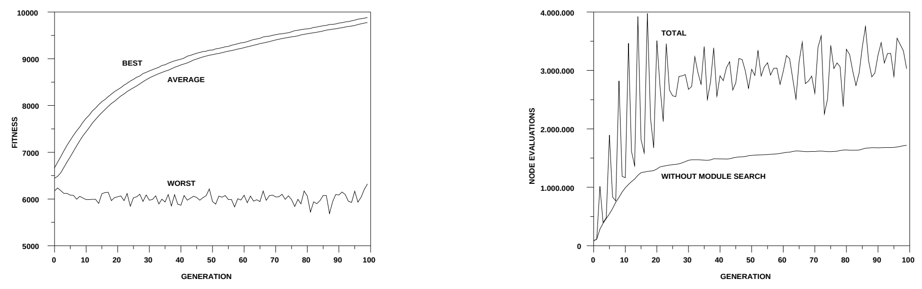


Figure 1.3: Even-7-parity hGP minor. 50 runs. Left: best, average, and worst fitness over time (measured in generation). Right: node evaluations per generation. Lower curve shows the node evaluation needed only for fitness evaluation. The upper curve shows the node evaluation needed for fitness evaluation and module search. The area between the upper and the lower curve represents the additional effort which is spent for searching good modules.

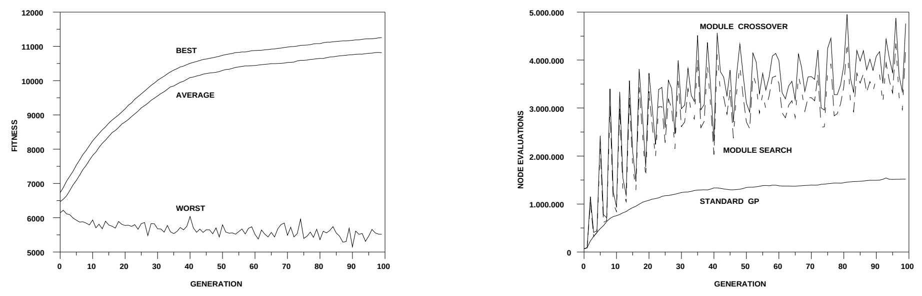


Figure 1.4: Even-7-parity hGP. 50 runs. Same as above.

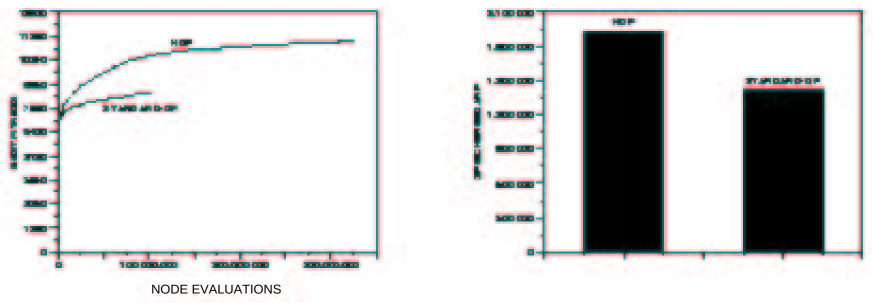


Figure 1.5: Even-7-Parity, GP vs hGP. 30 runs each. Left: Fitness progress over number of node evaluations. Right: Memory consumption compared between standard GP and hGP.

ness the nodes evaluated per generation is depicted in the right figures. It can be seen that hGPminor outperforms standard GP and hGP outperforms hGPminor. The number of nodes evaluated per generation increases which reflects the fact that average individual length is growing. The growing process is bounded because a parsimony pressure is activated. Note, that the pasimony pressure is very weak. 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. 1.5 time is now measured in node evaluations. For all 6 test problems (5 not shown here) hGP outperforms standard GP [4]. The performance gain depends on the problem. Figure 1.5 compares 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 considerably smaller. Even-7 parity is an exception from this rule.

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.

Bibliography

[1] Abelson, A., and G. Sussmann, *Structure and Interpretation of Computer Programs*, MIT Press Cambridge, MA (1985).

[2] Angeline, P. J., and J. B. Pollack, "The evolutionary induction of subroutines", *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, Lawrence Erlbaum (1992).

[3] Banscherus, D., "Hierarchische Genetische Programmierung mit lokalen Modulen" (1998), Diploma Thesis.

[4] Banzhaf, W., D. Banscherus, and P. Dittrich, "Hierarchical genetic programming using local modules", Technical Report *50/98*, University of Dortmund, Dortmund, Germany (1998).

[5] Banzhaf, W., P. Nordin, R Keller, and F. Francone, *Genetic Programming — An Introduction*, dpunkt/Morgan Kaufmann Heidelberg/San Francisco (1998).

[6] Fogel, D., *Evolutionary Computation*, IEEE Press Piscataway, NY (1995).

[7] Friedberg, E., G. Walker, and W. Siede, *DNA Repair and Mutagenesis*, ASM Press New York (1995).

[8] Gruau, F., "Genetic synthesis of modular neural networks", *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93* (University of Illinois at Urbana-Champaign,) (S. Forrest ed.), Morgan Kaufmann (17-21 July 1993), 318–325.

[9] Koza, J., *Genetic Programming – On the Programming of Computers by Means of Natural Selection*, MIT Press Cambridge, MA (1992).

[10] Koza, J., *Genetic Programming II*, MIT Press Cambridge, MA (1994).

[11] Koza, J., D. Andre, F. Bennett, and A. Keane, *Genetic Programming III*, Morgan Kaufmann San Francisco, CA (1999).

[12] Koza, J., F. Bennett III, D. Andre, M. Keane, and F. Dunlap, "Automated synthesis of analog electrical circuits by means of genetic programming", *IEEE Transactions on Evolutionary Computation* **1**, 2 (July 1997), 109–128.

[13] Olmer, M., W. Banzhaf, and P. Nordin, "Evolving real-time behavior modules for a real robot with genetic programming", *Proceedings of the international symposium on robotics and manufacturing* (Montpellier, France,), (May 1996).

[14] Osawa, S., *Evolution of the Genetic Code*, Oxford University Press Oxford (1995).

[15] Rosca, J., "An analysis of hierarchical genetic programming", Technical Report *566*, University of Rochester, Rochester, NY, USA (1995).

[16] Rosca, J., and D. H. Ballard, "Evolution-based discovery of hierarchical behaviors", *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, AAAI / The MIT Press (1996).

[17] Rosca, J. P., and D. H. Ballard, "Learning by adapting representations in genetic programming", *Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Orlando, Florida, USA* (Orlando, Florida, USA,), IEEE Press (27-29 June 1994).

[18] Rosen, R., *Life Itself*, Columbia University Press New York (1995).