Empirical Analysis of Different Levels of Meta-Evolution

Wolfgang Kantschik, Peter Dittrich, Markus Brameier, and Wolfgang Banzhaf

Department of Computer Science
University of Dortmund
Germany
wkantsch/dittrich/brameier/banzhaf@LS11.cs.uni-dortmund.de
http://ls11-www.cs.uni-dortmund.de

Abstract-

In this contribution we analyze different levels of meta-evolution using a graph-based GP system. The system allows to represent individuals of the search space and genetic variation operators in a coherent way as graph-programs differing only in the operator set. Seven variants of meta-evolution are tested on three real-world classification problems. The most complex variant consists of three meta-levels where graph-programs on meta-level 1 recombine individuals of the search space (base level), graph-programs on meta-level 2 recombine programs on meta-level 1, and programs on meta-level 3 recombine programs on meta-level 2 and themselves. The emperical results shows that the use of meta levels is advantageous.

Keywords: genetic programming, graph GP, self-adaptation of genetic operators, classification

1 Introduction

In the domain of evolutionary algorithms (EA), there is a long tradition of adaptive genetic operators. A common and well established method is the self-adaptation of strategy parameter, e.g., the global frequency of operator applications [6] or adaptation of the mutation variance in ES [22], EP [10], or GA [3]. In addition there are approaches which dynamically adjust the global interpretation of the representation based on heuristics [15, 23, 26]. There are also methods which allow adaptation of crossover operators by adjusting the probability that a position is chosen as a crossover point [20, 21]. This approach has also been successfully applied to GP [2, 11].

In GP, there is also an implicit adaptation of variation by neutral variation of the genotype. This usually happens implicitly in GP when introns appear that change e.g. the probability that a useful region is hit by recombination. There are also a variety of methods which explicitly manipulate the genotype like ADFs [13], adaptive representations [19], automatic library generation [1], explicitly defined introns [27], or module acquisition techniques [4, 11].

The methods of genetic programming, however, can be applied themselves as adaptions mechanisms if the variation operators are represented as programs [12, 24, 25]. The focus of our contribution is this meta-evolution of recombination-like variation operators. To this end, several schemes of how adaptive operators may be adapted are analyzed empirically. The adaptation of the adaptation operators can be achieved by adding additional evolutionary levels or by recursively applying the variation operators onto themselves [7, 14]. Here, this is operationalized by expressing genetic operators as graph programs that may undergo their own evolution, using the same methods in a hierarchical and recursive fashion. Before describing the seven variants of meta-evolution that we examine the following section introduces briefly our graph GP system which is based on an approach by Teller [24, 25].

1.1 Graph GP

The representations of programs used for GP can be classified by their underlying structure into three major groups: (1) tree-based [13], (2) linear-based [5, 16], and (3) graph-based [17, 25] representations.

In this paper we use Teller's *graph-based* GP. In this form of *graph-based* GP each program p is represented by a directed graph of N_p nodes. Each node can have up to N_p outgoing edges. Each node in the program has two parts, *action* and *branching decision*. The *action* part is either a constant or a function which will be executed when the node is reached during the interpretation of the program. The environment of a program consists of an indexed memory and a stack, both of which are used to transfer data among the nodes. An action function could therefore get its inputs from the stack and could push its output back onto the stack. After the action of a node is executed, an outgoing edge is selected according to the branching decision. This decision is made by a *branching function* which determines the edge to the next node, while using the information held on the top of the stack, in memory or in the *branching constant* of each node. Hence, not all nodes of a graph are necessarily visited during an interpretation.

Each program has two special nodes, a *start* and a *stop node*. The start node is always the first node to be executed when the interpretation of a program begins. After the stop node is reached, its action is executed and the program halts. Since the graph structure inherently allows loops and recursion, it is possible that the stop node is never reached during the interpretation. In order to avoid that a program runs forever it is terminated after a certain *time threshold* is reached. In our system the time threshold is implemented as a fixed maximum number of nodes which can be executed during the interpretation.

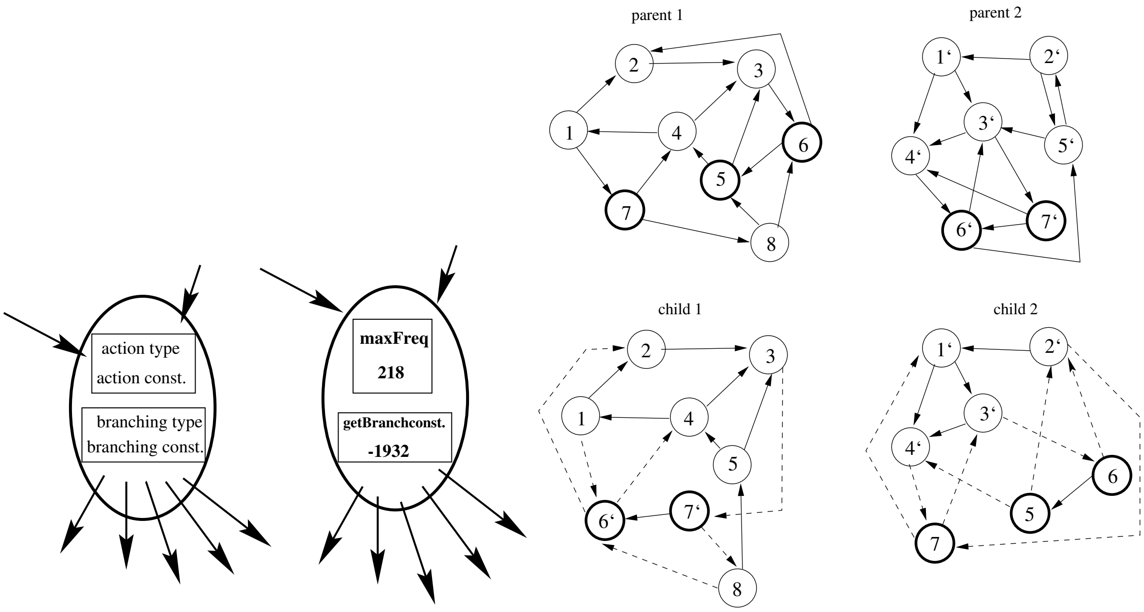

Figure 1: **Left:** The structure of a node in a graph-based GP program and an example node. **Right:** Crossover-operation example of two graph-based programs.

1.2 Recombination of Graph-based Programs

The crossover operation combines the genetic material of two parent programs by swapping certain program parts. Each node of a parent p is label-ed by a fixed index $i \in \{1, \ldots, N_p\}$. The following algorithm for the recombination of graphs is applied for recombination [24, 25]:

1. Mark some nodes in both parents which will be exchanged.
   (Here, this operation will be performed either by a random selection of nodes or by a "meta^i" operator to be explained below.)
2. Label all edges *external* which are connecting marked nodes with unmarked nodes and all edges which are connecting unmarked nodes with marked nodes.
3. Replace the nodes of a parent by the marked nodes of the other parent. A marked node with index i replaces a node with the same index in the other parent. If the target parent p does not contain a node with index i, then the node gets a new index $N_p + 1$ and will be added to the parent p.
4. Modify all *external edges* in a parent so that they point to randomly selected nodes of the same parent which have not been exchanged.

The method assures that all edges are connected in the two child graphs and that valid graphs are generated. Figure 1 shows an example of this crossover method.

2 Variants of Meta-Evolution

To explore different variants of meta-evolution our system consists of four different levels called task level, meta-1 level, meta-2 level and meta-3 level. Each level consists of a population of graph-programs where programs on the task level should solve the desired problem. Programs on meta levels are variation operators. The following variants are empirically investigated (Fig. 2):
Variant (a) "random": This is the conventional GP approach, where individuals are recombined by exchanging randomly chosen sub-components. There are demes of individuals that should solve the different classes of the classification problems. The individuals are called *task programs* to distinguish them from individuals of the higher levels, explained below.
Variant (b) "meta-1 random": In this variant a second population of GP programs exists. These individuals are called *meta-1 operators* and their population *meta-1 level*. Task programs are recombined by meta-1 operators [24]. Meta-1 operators, in turn, are recombined by a random recombination as in variant (a).
Variant (c) "meta-1 self": Like variant (b) but meta-1 operators are recombined by themselves.
Variant (d) "meta-2 random": Like variant (b) but meta-1 operators are recombined by meta-2 operators. Meta-2 operators form a third population (*meta-2 level*) and are recombined by a random recombination operator.
Variant (e) "meta-2 self": Like variant (b) but meta-1 operators are recombined by meta-2 operators. Meta-2 operators form a third population (*meta-2 level*) and are recombined by themselves.

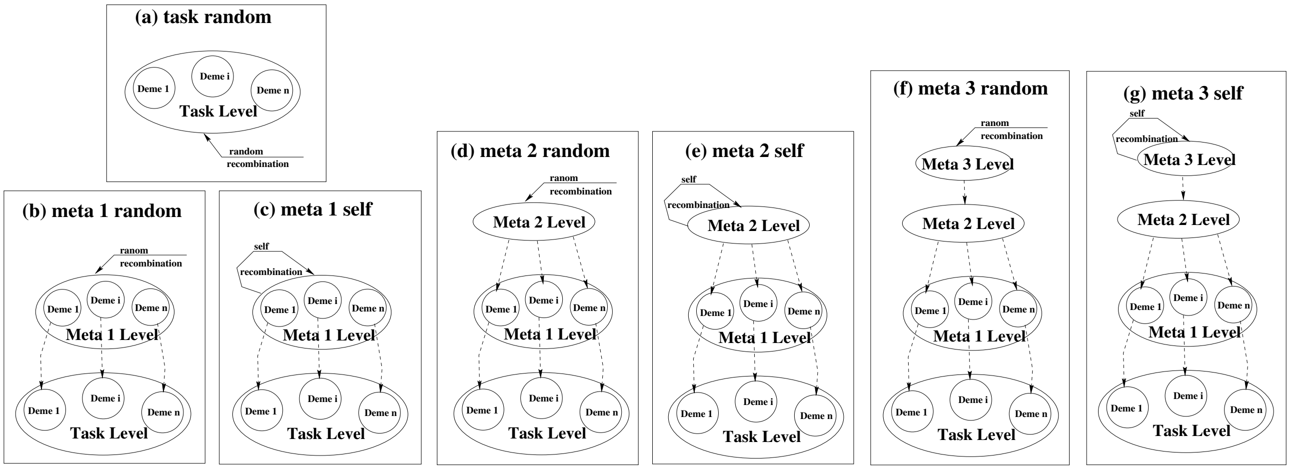

Figure 2: The system structure of the different recombination variants. Variant (a) a conventional GP approach, variant (b) is using the task level to recombine programs of the task level and the task level is recombined by a random recombination. Variant (c) is like variant (b) but meta-1 operators are recombined by themselves. Variant (d) is using the meta-2 level to recombine programs of meta-1 level and the operators of meta-2 level are recombined by a random recombination. Variant (e) is like variant (d) but the operators of the meta-3 level are recombined by themselves. Variant (f) is using meta-level 3 to recombine programs on meta-2 level and the operators of meta-level 3 are recombined by a random recombination. Variant (g) is like variant (f) but the operators on meta-3 level are recombined by themselves.

Variant (f) "meta-3 random": Like variant (d) but meta-2 operators are recombined by meta-3 operators. Meta-3 operators form a fourth population (*meta-3 level*) and are recombined by a random recombination operator.
Variant (g) "meta-3 self": Like variant (b) but meta-1 operators are recombined by meta-3 operators. Meta-3 operators form a fourth population (*meta-3 level*) and are recombined by themselves.

The following sections describe the structure of the evolutionary system, and the different levels in more detail.

2.1 The Task Level

The task level holds the population of task programs which should solve the desired test problem (here a classification problem). On the task and meta-1 level, the population is subdivided into sub-populations called *demes*. The number of demes depends on the number of classes, which have to be distinguished. Four demes are needed for the speaker identification problem (Sec. 3.1), two demes for the cancer and diabetes problem, and four demes for the gene problem 3.2. The task programs of deme i are responsible to classify the input data of class i correctly. A task program p^i_{task} belonging to deme i represents a mapping $p_{task} : Input \rightarrow [min, max]$ where $Input$ represents the set of input data which should be classified. The output is a number interpreted as a confidence value. A high value value of $p^i_{task}(x)$ is interpreted such that $x \in Input$ belongs probably to class i.

2.1.1 Variation and Selection on Task Level

The selection method on the task level is a tournament-strategy [5]. The variation method depends on the variant: **Variant (a)** uses random recombination and applies random mutation of a maximum 5 nodes of the program after recombination. The recombination rate and mutation rate is 100 %, this means that each program of the task level is recombined and mutated during one generation. **Variants (b)-(g)** use recombination by randomly chosen meta-1 operators from the meta-1 level. Mutation is only performed by an explicit mutate instruction as part of the meta-1 operator set in meta-1 programs.

2.2 Meta Levels

Meta-operators[1] should enable the GP system to find a good and suitable recombination method automatically. Therefore, a meta-operator p_{meta} represents a mapping $p_{meta} : P \times P \rightarrow P \times P$ where P is the set of all programs (task programs or meta-operators).

2.2.1 Fitness Function on Meta Levels

The goal of a meta-n operator with respect to the underlying level $n-1$ population is to maximize the fitness of meta-$(n-1)$ programs[2] . A meta-operator is tested by allowing it to actually perform a recombination of program on the underlying level. Its

[1] A Meta-1 operator is called "operator" by Edmonds [8] and smart operator by Teller [24]

[2] or task programs in case $n = 1$

fitness value is a function of the relative fitness of the programs it recombines (parents) and the fitness of programs it produces as descendants (children). To compute this fitness in a generation-based evolutionary algorithm the relative fitness increase a meta-operator is able to cause on programs of the underlying level during a generation is accumulated by using measure Ω. The following algorithm describes the fitness calculation for meta-1 level which is equivalent on all meta levels. The algorithm represents a loop of one generation during which λ offsprings are generated.

1. Reset counters:
   $\forall p \in P : \Omega(p) \leftarrow 0, m(p) \leftarrow 0, n(p) \leftarrow 0.$
2. Select two parents $p_{task}^{(p1)}, p_{task}^{(p2)}$ from the task level and a meta-1 operator p_{meta} from the meta-1 level, randomly.
3. Create two offsprings by applying the meta-1 operator on parents:
   $$(p_{task}^{(c1)}, p_{task}^{(c2)}) = p_{meta}(p_{task}^{(p1)}, p_{task}^{(p2)}).$$
4. FOR $j = 1$ TO 2 DO
   (a) Let $n(p_{meta}) \leftarrow n(p_{meta}) + 1$
   (b) Let $f_{cj} = Fit_{task}(p_{task}^{(cj)})$ and $f_{pj} = Fit_{task}(p_{task}^{(pj)})$ be the fitness of a child and its corresponding parent, respectively.
   (c) If the childs fitness f_{cj} is better than the parents fitness f_{pj} then let
   $$\begin{aligned} \Omega(p_{meta}) &\leftarrow \Omega(p_{meta}) + \frac{f_{cj} + f_{max}}{f_{pj} + f_{max}} - 1, \\ m(p_{meta}) &\leftarrow m(p_{meta}) + 1. \end{aligned}$$
   where f_{max} is the maximal fitness a program can reach,
5. GOTO 2 UNTIL λ task programs are created to form the next generation.
6. The fitness of a meta-1 operator is defined by
   $$Fit_{meta}(p_{meta}) = \frac{m(p_{meta})}{n(p_{meta})} * \Omega(p_{meta}).$$

This means that a meta-1 operator is good, if the children (at least one) have a better fitness than the parents.

2.2.2 Representation and Operator Set on Meta Levels

A meta operator recombines two given programs by marking some nodes in both parents according to step 1 of the recombination algorithm in Sec. 1.2. To perform this task the meta-operator needs the ability to examine its input programs in sufficient detail. Therefore we provide the individuals with *special action functions*, with these functions the individual can examine the input individuals.

During the execution of a meta operator the environment contains an additional element: the *current node*, this is the program node the meta operator currently works with. The meta operator executes its graph-program at first on parent $p_{task}^{(p1)}$ and then independently on $p_{task}^{(p2)}$. After the meta operator has been executed, the new child programs will be created by exchanging the marked nodes according to the algorithm in Sec. 1.2. If a parent does not have a marked node, the meta-1 operator receives fitness 0 and a random crossover is executed.

The meta operators used for this study also mark a subset of nodes to be mutated after the recombination. So the recombination process of a meta operator is a combination of a crossover and a mutation operation. The selection method on all meta levels is tournament selection. The variation method depends on the variant described in Sec. 2.

3 Test Problems

We use different classification problems as test problems. One test problem is a speaker identification problem, and the other classification problems are chosen form the proben1 benchmark set [18].

3.1 Speaker Identification

The speaker identification problem considered in this study is to identify one person out of a set of four persons based on speech samples [9].

The raw sound data was sampled at 22 kHz in 16 bit format. A fast Fourier transformation has been done on a 20 msec window, which was weighted using a Hamming window. Windows were overlapping by 50 %. A spectral vector of dimension 32 was computed out of these FFT spectral vectors by using a triangle filter. The spectral vectors for the different word groups and speakers were received by the task programs as inputs to identify whether a given input (a word group of one speaker) belongs to the specific class (speaker) or not. The input data for one identification task consist of two different words form each class, i.e., the task program has to identify a speaker based on a speech sample of less than 2 seconds.

3.1.1 Fitness Function

The fitness function uses the return values of a input set to determine the fitness value of an individual. The return value of an individual is a number between $min = -10000$ and $max = 10000$. The normalised return value is interpreted as a measure of probability. If the return value is high and the individual is associated with class i, then the input sample is identified as belonging to class i. By combining the identification result of programs associated with different classes it is possible to identify the speaker for a given input.

The fitness $Fit(p_{task})$ of a task program p_{task} associated with class i on the fitness cases C, is computed as

$$Fit(p_{task}) = \sum_{e \in \mathcal{C}_i} ((n_C - 1) * r(p_{task}, e)) - \sum_{e \notin \mathcal{C}_i} r(p_{task}, e) \tag{1}$$

where $r(p, e)$ is the return value of program p executed with input e, $\mathcal{C}_i \subset C$ is the subset of the fitness cases containing only samples class i, and n_C is the number of classes.

3.1.2 Operator Set on Task Level

On the task level programs need the ability to examine input data (spectral vectors) in sufficient detail in order to perform their task. Therefore they use various functions which operate directly on the input vectors. These function can read the values at a special position in the input sample, compare two values of the input data, or calculate the average or difference of some input values. The programs have no opportunity to store a vector of the input data to compare it later to other input. In other words, programs have to identify a speaker without the use of reference vectors. This distinguishes the method from classical solutions for the speaker identification problem. Stack and indexed memory only store one-dimensional real values during the execution of program. Task programs use a action function set which consists of arithmetic functions, comparison functions, and functions to examine the input data like reading the value of a given frequency, or calculating the average value of a vector and so on.

3.2 Proben1 Benchmark Set

The proben1 benchmark set [18] contains datasets to be used for neural network training. Proben1 contains 15 data sets form 12 different domains. In this contribution we chose three classification problems out of the data set, the diabetes data set, the cancer data set, and the gene data set.

3.2.1 Fitness Function

The fitness function uses the return values of a input set to determine the classification rate which is used as the fitness value.

The return value of an individual is a number between $min = -10000$ and $max = 10000$. The normalized return value is interpreted as a measure of probability. If the return value is high and the individual is associated with class i, then the input sample is identified as belonging to class i. By combining the identification result of programs associated with different classes it is possible to identify the speaker for a given input.

The fitness $Fit(p_{task})$ of a task program p_{task} associated with class i on the fitness cases C, is computed as

$$Fit(p_{task}) = \begin{cases} Fit(p_{task}) + 1 & \text{, if } e \in \mathcal{C}_i \text{ and } r(p_{task}, e) > 0. \\ Fit(p_{task}) + 1 & \text{, if } e \notin \mathcal{C}_i \text{ and } r(p_{task}, e) < 0. \end{cases} \tag{2}$$

where $r(p, e)$ is the return value of program p executed with input e, $\mathcal{C}_i \subset C$ is the subset of the fitness cases containing only samples class i, and n_C is the number of classes.

3.2.2 Operator Set on Task Level

On the task level programs the programs use various functions which operate directly on the input vector. Therefore the current input vector is stored in a register set.

The programs can only read the registers and have no opportunity to store a vector of the input data for later comparison it later to other input. Stack and indexed memory only store one-dimensional real values during the execution of program. Task programs use a action function set which consists of arithmetic functions, comparison functions, and functions to examine the input data.

4 Results

In this section we describe the effects of the seven variants (a) *task random*, (b) *meta-1 random*, (c) *meta-1 self*, (d) *meta-2 random*, (e) *meta-2 self*, (f) *meta-3 random*, and (g) *meta-3 self* recombination.

For the speaker problem the task level contains 192 programs in each deme and the maximum allowed number of nodes is set to 300 nodes. The meta-1 population contains 96 operators in each deme and the maximum number of allowed nodes is set to 300 nodes. The meta-2 population contains 48 operators with the same structure as the meta-1 operators. The meta-3 population contains 24 operators with the same structure as the meta-2 operators. In each generation the programs are tested with 6 randomly chosen examples from the training set (stochastic sampling).

For the other three problems the task level contains 224 programs in each deme and the maximum allowed number of nodes is set to 300 nodes. The meta-1 population contains 112 operators in each deme and the maximum number of allowed nodes is set to 300 nodes. The meta-2 population contains 56 operators with the same structure as the meta-1 operators. The meta-3 population contains 28 operators with the same structure as the meta-2 operators. In each generation the programs are tested with 100 randomly chosen examples from the training set (stochastic sampling).

On the gene problem nearly all 20 runs reached 100 percent correct classification on the fitness cases in a few generations so that they can not be used to differentiate the different meta variants.

4.1 Fitness

Figure 3 shows the progression of the fitness values. The figure shows the advantage of *meta-1 random* and *meta-2 self* recombination over *task random*. The results also confirm for the speaker identification problem, that self-recombination on the meta-1 level has a lower performance which complies with the results in [12]. For the cancer and diabetes problem there is no significant difference in the performance of all meta variants (b)-(g). Despite the high standard deviation variant (c) meta-1 self has always a lower performance than variant (e) meta-2 self. This may be taken as an indication that self-application of operators on meta-level 2 is preferable to self-application on meta-level 1.

There is no evidence that a third meta level increases performance. For the speaker problem a performance decrease can be observed.

The experiments show that the meta evolutionary methods are performing better than random crossover. Among them, the meta-1 self-crossover method seems to have overall the worst performance. For the speaker identification problem it is even worse than the random recombination method. The meta-2 self variant seems to have the best performance. Therefore a system with two meta levels and self-recombination at the second meta level (meta-2 self) is recommended from these experiments.

	deme 0	σ	deme 1	σ	deme 2	σ	deme 3	σ
rand	213000	5300	208000	4900	215000	5300	211000	5300
meta-1 rand	218000	5500	218000	5500	218000	5500	218000	5500
meta-1 self	212000	5100	212000	5100	216000	5300	215000	5100
meta-2 rand	217000	5600	222000	6300	217000	5600	217000	5600
meta-2 self	244000	17000	243000	17000	240000	15000	236000	14000
meta-3 rand	218000	5100	217000	5100	220000	6000	217000	5200
meta-3 self	218000	5100	218000	5100	218000	5100	218000	5100

Table 1: Best fitness values of each deme in the last generation (100) averaged over 25 runs including standard deviation for the speaker problem.

	deme 0	σ	deme 1	σ
rand	67.8	0.1	68.2	0.1
meta-1 rand	74.5	0.7	73.6	0.6
meta-1 self	74	0.6	73.1	0.5
meta-2 rand	75.0	0.5	74.3	0.4
meta-2 self	75.5	0.7	74.2	0.4
meta-3 rand	74.3	0.4	74.3	0.5
meta-3 self	74.1	0.2	73.8	0.2

	deme 0	σ	deme 1	σ
rand	65.9	0.2	66.2	0.2
meta-1 rand	71.1	0.4	71	0.5
meta-1 self	71.2	0.3	71.6	0.4
meta-2 rand	72.2	0.2	71.8	0.2
meta-2 self	72	0.2	72.3	0.3
meta-3 rand	72.8	0.4	72.5	0.4
meta-3 self	72.1	0.3	71.9	0.2

Table 2: Best fitness values of each deme in the last generation (100) averaged over 55 runs each and standard deviation σ. **Left:** Diabetes problem. **Right:** Cancer problem.

5 Summary and Outlook

We tested the performance of meta-recombination with different meta levels to apply GP to find a better recombination scheme. We have shown that it is possible to create a GP system which does not use a fixed recombination operator, and that such a system can create individuals with better fitness.

In our experiments *self* recombination at meta-level 1 has the smallest effect on the evolutionary process and for the speaker identification problem it reduces performance. *Self* recombination at the meta-2 and meta-3 level works. All meta-recombination variants (except meta-1 self for the speaker problem) could find a recombination scheme which is better than

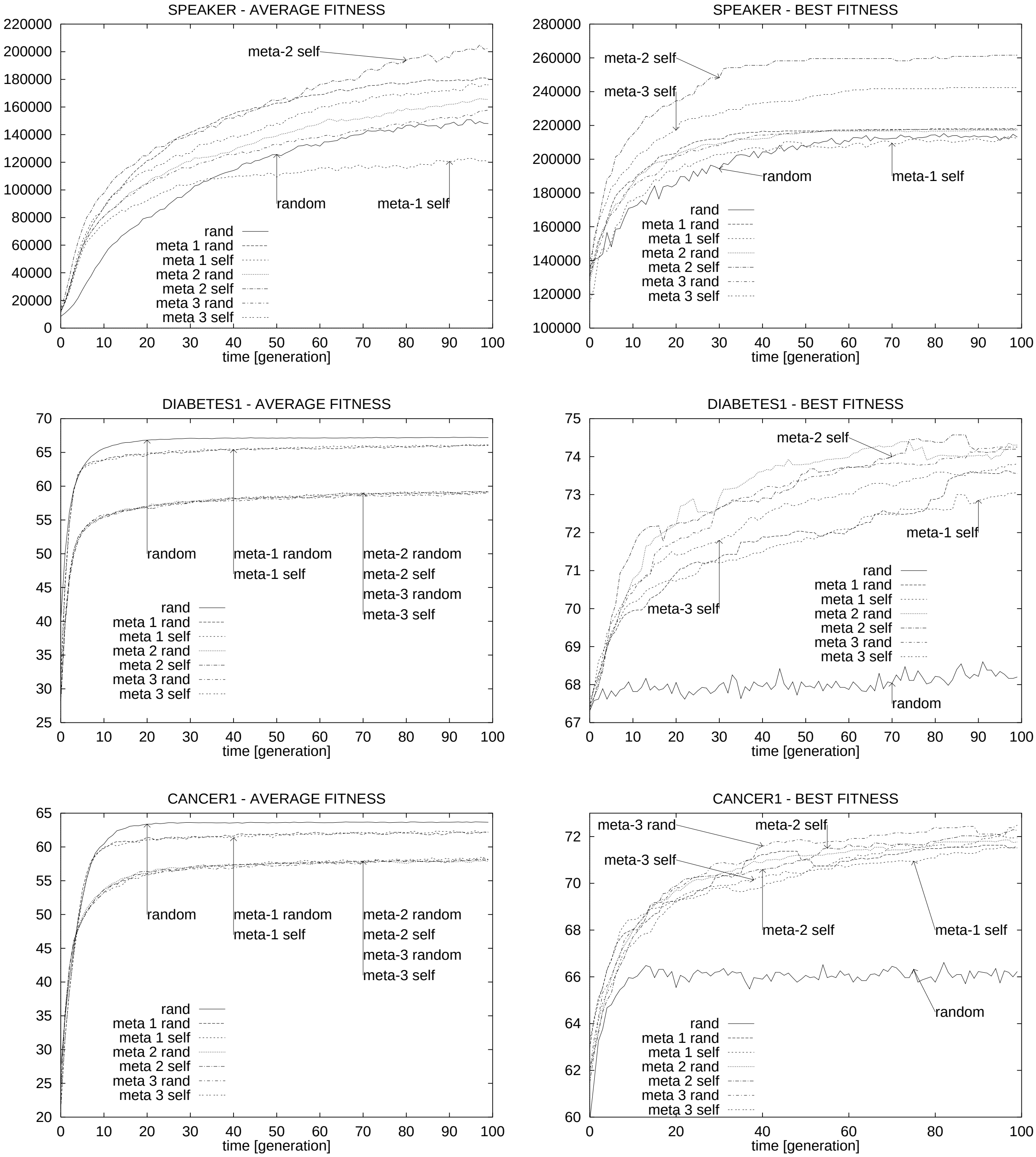

Figure 3: Results for speaker problem, above (25 runs, $\sigma = 30000$), cancer problem, middle (55 runs, $\sigma = 1.5$), and diabetes problem, below (55 runs, $\sigma = 3$).

random recombination. The variant meta-2 self seems to have a good performance for all test problems. A third meta-level seems not to be useful.

To say whether these are general phenomena for genetic programming more experiments have to be run on a variety of test problems and on different representations for the task programs and meta-i operators.

ACKNOWLEDGEMENT

Support has been provided by the DFG (Deutsche Forschungsgemeinschaft), under grant Ba 1042/5-2 and under grant B2 in the Sonderforschungsbereich SFB 531.

Bibliography

[1] P. J. Angeline and J. B. Pollack. Coevolving high-level representations. In C. G. Langton, editor, *Artificial Life III*, pages 55–71, Reading, MA, 1994. Addison-Wesley.

[2] P.J. Angeline. Two self-adaptive crossover operations for genetic programming. In P.J. Angeline and K.E. Kinnear, Jr., editors, *Advances in Genetic Programming II*, pages 89–110. MIT Press, Cambridge, MA, 1996.

[3] Th. Bäck. Self-Adaptation in Genetic Algorithms. In F. J. Varela and P. Bourgine, editors, *Proceedings of the First European Conference on Artificial Life*, pages 263–271. The MIT Press, Cambridge, MA, 1992.

[4] W. Banzhaf, D. Banscherus, and P. Dittrich. Hierarchical genetic programming using local modules. In *Proc. Intl. Conference on Complex Systems, Nashua, NH, (in press)*, 1998.

[5] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco und dpunkt verlag, Heidelberg, 1998.

[6] Lawrence Davis. Adapting operator probabilities in genetic algorithms. In J. David Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 61–69, George Mason University, June 1989. Morgan Kaufmann.

[7] Peter Dittrich and Wolfgang Banzhaf. Self-evolution in a constructive binary string system. *Artificial Life*, 4(2):203–220, 1998.

[8] B. Edmonds. Meta-genetic programming: Co-evolving the operators of variation. CPM Report 98-32, Manchester Metropolitan University, 1998.

[9] R.A. Finan, A.T. Sapeluk, and R.I. Damper. VQ score normalisation for text-dependent and text-independent speaker recognition. In *Audio- and Video-based Biometric Person Authentication*, pages 211–218. First International Conference, AVBPA'97, 1997.

[10] D. B. Fogel, L. J. Fogel, and J. W. Atmar. Meta-evolutionary programming. In R. R. Chen, editor, *Proceedings of 25th Asilomar Conference on Signals, Systems and Computers*, pages 540–545, Pacific Grove, California, 1991.

[11] Hitoshi Iba and Hugo de Garis. Extending genetic programming with recombinative guidance. In P.J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming II*, chapter 4, pages 69–88. MIT Press, Cambridge, MA, 1996.

[12] W. Kantschik, P. Dittrich, M. Brameier, and W. Banzhaf. Meta-evolution in graph GP. In *Second European Workshop on Genetic Programming, Goteborg, May, (accepted)*, 1999.

[13] J. Koza. *Genetic Programming*. MIT Press, Cambridge, MA, 1992.

[14] John R. Koza. Spontaneous emergence of self-replicating and evolutionarily self-improving computer programs. In C. G. Langton, editor, *Artificial Life III*, pages 225–262. Addison-Wesley, Reading, MA, 1994.

[15] Keith Mathias and Darrell Whitley. Remapping hyperspace during genetic search: Canonical delta folding. In L. Darril Whitley, editor, *Proceedings of the Second Workshop on Foundations of Genetic Algorithms*, pages 167–186, San Mateo, July 26– 29 1993. Morgan Kaufmann.

[16] J. P. Nordin. A compiling genetic programming system that directly manipulates the machinecode. In Jr. K. Kinnear, editor, *Advances in Genetic Programming*. Cambridge, MIT Press, 1994.

[17] Riccardo Poli. Evolution of graph-like programs with parallel distributed genetic programming. In Thomas Back, editor, *Genetic Algorithms: Proceedings of the Seventh International Conference*, pages 346–353, Michigan State University, East Lansing, MI, USA, 19-23 July 1997. Morgan Kaufmann.

[18] L. Prechelt. Proben1 — a set of neural network benchmark problems and benchmarking rules. Technical report, University of Karlsruhe, 1994.

[19] J. P. Rosca and D. 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.

[20] R. Rosenberg. *Simulation of Genetic Populations with Biochemical Properties*. PhD thesis, University of Michigan, 1967.

[21] J D Schaffer and A Morishima. An adaptive crossover distribution mechanism for genetic algorithms. In *Proc of the 2nd Int. Conf. on Genetic Algorithms and Their Applications*, pages 36–40, 1987. SCHAFFER87.

[22] H.-P. Schwefel. *Evolution and Optimum Seeking*. Wiley, New York, 1996.

[23] Craig G. Shaefer. The ARGOT strategy: adaptive representation genetic optimizer technique. In John J. Grefenstette, editor, *Proceedings of the 2nd International Conference on Genetic Algorithms and their Applications*, pages 50–58, Cambridge, MA, July 1987. Lawrence Erlbaum Associates.

[24] A. Teller. Evolving programmers: The co-evolution of intelligent recombination operators. In P. Angeline and K. Kinnear, editors, *Advances in Genetic Programming II*. MIT Press, 1996.

[25] A. Teller and M. Veloso. Pado: A new learning architecture for object recognition. In *Symbolic Visual Learning*, pages 81 –116. Oxford University Press, 1996.

[26] D. Whitley, K. Mathias, and P. Fitzhorn. Delta coding: An iterative search strategy for genetic algorithms,. In Rick Belew and Lashon Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 77–84, San Mateo, CA, 1991. Morgan Kaufman.

[27] Annie S. Wu and Robert K. Lindsay. Empirical studies of the genetic algorithm with non-coding segments. *Evolutionary Computation*, 3(2), 1995.