Meta-Evolution in Graph GP

Wolfgang Kantschik[1,2] Peter Dittrich[1] Markus Brameier[1] and Wolfgang Banzhaf[1,2]

[1] Dept. of Computer Science, University of Dortmund, Dortmund, Germany
[2] Informatik Centrum Dortmund (ICD), Dortmund, Germany

Abstract. In this contribution we introduce the evolution of operators for Genetic Programming by means of Genetic Programming. Specifically, meta-evolution of recombination operators in graph-based GP is applied and compared to other methods for the variation of recombination operators in graph-based GP. We demonstrate that a straightforward application of recombination operators onto themselves does not work well. After introducing an additional level of recombination operators (the meta level) which are recombining a pool of recombination operators, even self-recombination on the additional becomes feasible. We show that the overall performance of this system is better than in other variants of graph GP. As a test problem we use speaker recognition.

1 Introduction to Graph GP

The representations of programs used in Genetic Programming can be classified by their underlying structure into three major groups: 1) tree-based [Koz92,Koz94], 2) linear-based [Nor94,BNKF98], and 3) graph-based [TV96] representations.

In *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

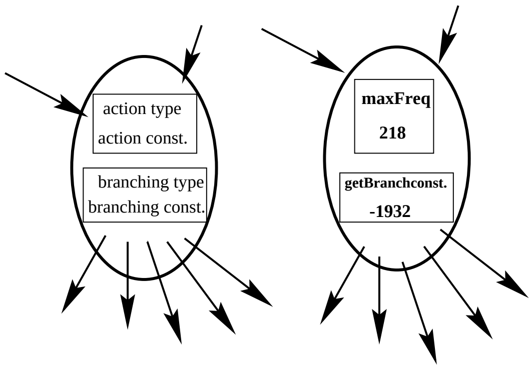

Fig. 1. The structure of a node in a graph-based GP program (left) and an example node (right).

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.

1.1 Recombination of Graph-based Programs

The crossover operation combines the genetic material of two parent programs by swapping certain program parts. The following algorithm for the recombination of graphs is applied for recombination (see also [Tel96]):

1. Divide each graph into two sets of nodes.
   (Here, this operation will be performed either by a random selection of nodes or by a "smart" operator to be explained below.)
2. Label all edges *internal* if they point to another node in the same set, or external otherwise.
3. Label nodes in each set as *output* if they are the source of an external edge and as *input* if they are the destination of an external edge.
4. Exchange one set from each graph.
5. Recombine so that all *external* edges in each fragment point to randomly selected *input* nodes in the other set.

The method assures that all edges are connected in the two child graphs and that valid graphs are generated. Figure 2 shows an example of this crossover method.

2 Levels of Evolution

The focus of this contribution is the meta-evolution of genetic recombination-like variation operators. This is done by expressing (recombination) operators as

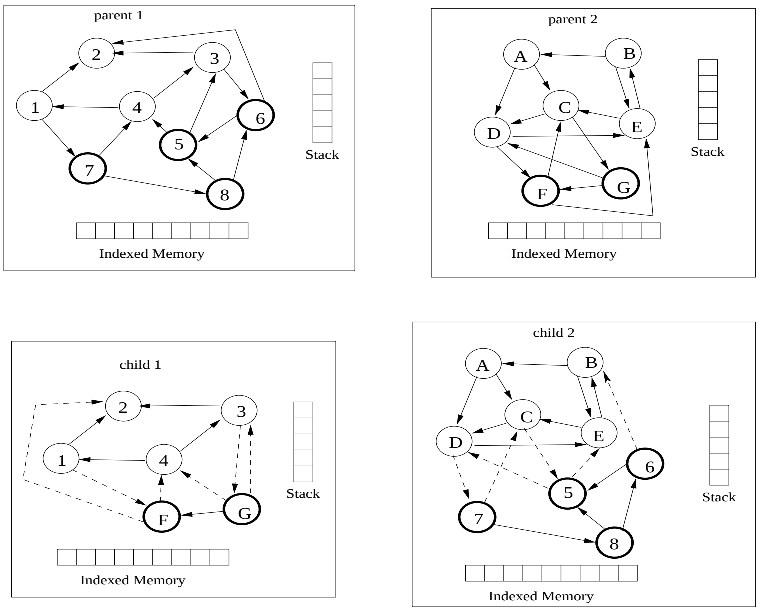

Fig. 2. Crossover-operation of two graph-based programs.

graph programs that may undergo their own evolution, possibly using the same methods. In order to compare different approaches we consider four variants (Fig. 3):

Variant (a) "task random": This is the conventional GP approach, where individuals are recombined by exchanging randomly chosen sub-components. There is only one population of individuals that should solve the desired task, here, speaker identification. Its individuals are called *task programs* to distinguish them from individuals of the higher level (smart or meta), explained below.

Variant (b) "smart random": In this variant a second population of GP programs exists. These individuals are called *smart operators* and their population *smart level*. Task programs are recombined by smart operators in a better than random way [Tel96]. smart operators, in turn, are recombined by a random recombination as in variant (a).

Variant (c) "smart self": Like variant (b) but smart operators are recombined by themselves.

Variant (d) "meta self": Like variant (b) but smart operators are recombined by meta operators. Meta operators form a third population (*meta level*) and are recombined by themselves.

The following sections describe the task, smart and meta level in more detail.

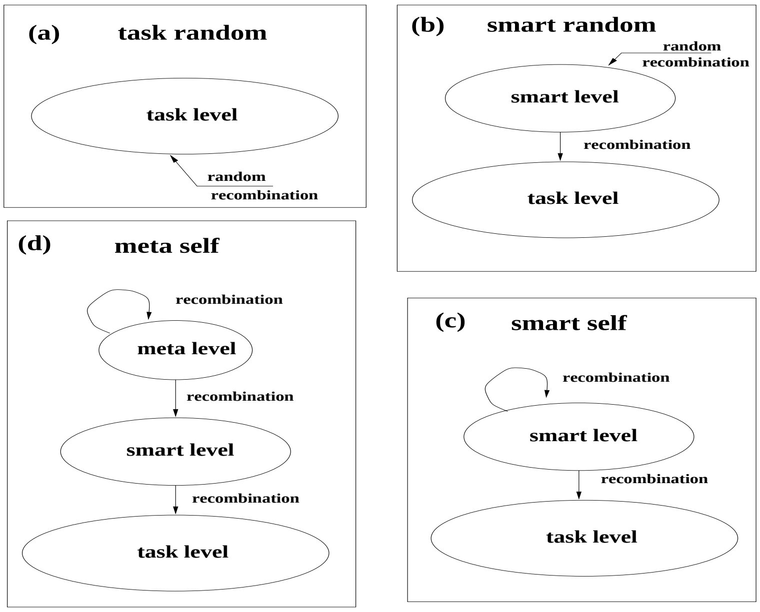

Fig. 3. The system structure of the different recombination variants. Variant (a) a conventional GP approach, variant (b) is using the smart level to recombine programs of the task level and the smart level is recombined by a random recombination. Variant (c) is like variant (b) but smart operators are recombined by themselves, and variant (d) is using the meta level to recombine programs of the the smart level and themselves.

3 The Task Level

The task level consists of a population of task programs which should solve the desired test problemtaken from speaker identification [RS86a,RS86b,FSD97]. Therefore, a task program p_{task} represents a mapping

$$p_{task} : Input \rightarrow [min, max] \quad (1)$$

where *Input* represents the set of spoken words (see below).

3.1 Fitness Function of Task Level

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

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 one to five different words form each class, i.e., the task program has to identify a speaker based on a speech sample of less than 5 seconds.

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

$$Fit(p_{task}) = \sum_{e \in \mathcal{C}_i} ((n_C - 1) * r(p_{task}, e)) - \sum_{e \notin \mathcal{C}_i} r(p_{task}, e) \qquad (2)$$

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 Representation and Operator Set of Task Level

On each level programs are represented as graphs. 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 the action function set shown in Tab. 1

3.3 Variation and Selection on Task Level

The selection method on the task level is a $(\mu + \lambda)$-strategy [Sch96]. The variation method depends on the variant: **Variant (a)** uses random recombination and applies random mutation of a maximum 5 % of the program nodes after recombination. **Variants (b)-(d)** use recombination by randomly chosen smart operators from the smart level. Mutation is only performed by an explicit mutate instruction as part of the smart operator set in smart programs (see below and Tab. 2).

function name	description of action function (task level)
$+, -, *, /$	arithmetic functions
$<, >, =$	comparison functions
readFrequency	read the value of a given frequency and spectral vector.
maxFreq	returns the frequency and value with the maximal value of a given spectral vector.
minFreq	returns the frequency and value with the minimal value of a given spectral vector.
interMaxFreq	returns the frame number and value of a given frequency with the maximal frequency value of 6 following spectral vectors.
interMinFreq	returns the frame number and value of a given frequency with the minimal frequency value of 6 following spectral vectors.
countFreq	returns the number of spectral vector which have a frequency value equal x.
frameAverage	returns the average frequency value of a given spectral vector.
interAverage	returns the average frequency value of a given frequency of a variable number of spectral vectors.
variance	returns the average variance value of a given spectral vector.
interVariance	returns the average variance value of a given frequency of a variable number of spectral vectors.
smallerFreq	returns the smaller value of two given frequency spectral vectors.
greaterFreq	returns the greater value of two given frequency spectral vectors.

Table 1. Action functions (operators, non-terminals) on task level.

4 The Smart Level

Smart operators[1] [Tel96] should enable the GP system to find a good and suitable recombination method automatically. Therefore, a smart operator p_{smart} represents a mapping

$$p_{smart} : P \times P \to P \times P \tag{3}$$

where P is the set of all programs (task or smart operators).

4.1 Fitness Function on Smart Level

The goal of the smart operators with respect to the task level population is to maximize the fitness of task programs. smart operators and task programs co-evolve. A smart operator is tested by allowing it to actually perform a recombination of task programs. Its fitness value is a function of the relative fitness of the task 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 smart operator is able to cause on task programs during a generation is accumulated by using Ω in the

[1] Simply called "operator" by Edmonds [Edm98].

following algorithm. The algorithm represents a loop of one generation during which λ task programs 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 smart operator p_{smart} from the smart level, randomly.
3. Create two offsprings by applying the smart operator on parents:
$$(p_{task}^{(c1)}, p_{task}^{(c2)}) = p_{smart}(p_{task}^{(p1)}, p_{task}^{(p2)}).$$
4. FOR $j = 1$ TO 2 DO
   (a) Let $n(p_{smart}) \leftarrow n(p_{smart}) + 1$
   (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 f_{cj} (child) is better than f_{pj} (parent) then let
$$\Omega(p_{smart}) \leftarrow \Omega(p_{smart}) + \frac{f_{cj} + f_{max}}{f_{pj} + f_{max}} - 1,$$
$$m(p_{smart}) \leftarrow m(p_{smart}) + 1.$$
   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 smart operator is defined by
$$Fit_{smart}(p_{smart}) = \frac{m(p_{smart})}{n(p_{smart})} * \Omega(p_{smart}).$$

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

4.2 Representation and Operator Set of Smart Operators

A smart operator recombines two given programs by creating subsets of nodes according to step 1 of the recombination algorithm in Sec. 1.1. To perform this task the smart operator needs the ability to examine its input programs in sufficient detail. Therefore we provide the *special action functions* shown in Tab. 2.

During the execution of a smart operator the environment contains three additional elements: (1) the *current node* this is the program node the smart operator currently works with. (2) *node set 1* in which the smart operator stores the nodes from the first parent $p_{task}^{(p1)}$ which will transferred into the second parent $p_{task}^{(p2)}$ to form the second child $p_{task}^{(c2)}$. (3) *node set 2* in which the smart operator stores the nodes from the second parent $p_{task}^{(p2)}$ which will transferred into the first

function name	description of action function (smart and meta level)
pickRand	Pick at random a new *current node* which is not member of the *set*.
pickNode	Picks a specific node to be the new current node.
pickChild	Picks node pointed to by an edge of the current node to be the new current node.
addCurrent	Adds current node to set.
addChild	Adds children of current node to set.
delCurrent	Deletes current node form set.
delChild	Deletes children of current node from set.
randSet	Makes set be a random set of nodes.
setSize	Returns size of set.
progSize	Returns size of program.
nodeAction	Returns action of current node.
nodeConst	Returns constant of current node.
nodeOutgrad	Returns outgrad of current node.
nodeBAction	Returns branching action of current node.
nodeBConst	Returns branching constant of current node.
mutate	Marks a specific node and a specific part of the node to be mutated after recombination.

Table 2. Action functions on smart and meta level.

parent $p_{task}^{(p1)}$ to form the first child $p_{task}^{(c1)}$. The smart operator executes its graph-program at first on parent $p_{task}^{(p1)}$ and then inependently on $p_{task}^{(p2)}$ to create *node set 1* and *node set 2*, respectively. After the smart operator has been executed and has created the node sets, the new child programs will be created by using the algorithm from section 1.1. If one of the two sets has size 0 the smart operator receives fitness 0 and a random crossover is executed.

The smart operators used for this study also mark a subset of nodes to be mutated after the recombination. So the recombination process of a smart operator is a combination of a crossover and a mutation operation.

4.3 Variation and Selection of Smart Operators

The selection method on the smart level is rank proportional. The variation method depends on the variant: **Variant (a)** does not use the smart level. **Variant (b)** uses random recombination for smart operators and applies random mutation of some nodes after recombination. **Variant (c)** uses recombination by choosing the best programs from the same level (smart level). Mutation is only performed by an explicit mutate instruction in the smart operators. **Variant (d)** uses recombination by randomly chosen programs from the meta level. Mutation is performed by an explicit mutate instruction in meta operators.

5 The Meta Level

Until now we have described a system with two levels, the *task level* and the *smart level*. The task programs at the task level are evolved to solve the given problem. Those at the smart level are evolved to perform recombination at the task level. One of the most obvious issues of the smart level is the evolution of smart operators.

New meta operators are introduced because self-recombination (variant (b)) does not work on the smart level (a result stated in [Tel96] and confirmed in Fig. 4 below). However, there is the chance that it works if a next level (the meta level) of evolution is introduced. Why should this be possible?

The reasoning might go as such: The goal of smart operators is to find a good recombination for programs on the task level and not for the smart level. Smart operators have a different structure than task programs. Therefore it is possible that a good smart operator for a task program has a low performance in recombining smart operators. Teller uses the *random* recombination for smart operators because empirically tests have shown that *self* recombination does not work at the smart level.

5.1 Fitness Function and Representation on Meta Level

To allow the self-recombination on the meta level (variant (d)) the meta level uses the same fitness function as the smart level (see Sec. 4.1). It also uses the same representation and operator set as the smart level (see Sec. 4.2).

5.2 Variation and Selection on Meta Level

The selection method on the meta level is rank proportional. The variation method depends on the variant: **Variants (a)-(c)** do not use the meta level. **Variant (d)** recombines meta operators by choosing the best programs from the same level (meta level) as recombination operators. Mutation is only performed by an explicit mutate instruction in the meta operators.

6 Test Problem Results

In this section we describe the effects of the four variants (a) *task random*, (b) *smart random*, (c) *smart self*, and (d) *meta self* recombination.

In this study the task level contains 400 programs and the maximum number of nodes for each program is 600. The smart population contains 100 operators and the maximum number of nodes is 300. The meta population contains 50 operators with the same structure as the smart operators. The fitness cases consist of 15 sound examples for each speaker. In each generation the programs are tested with 6 randomly choosen examples from each speaker.

The task level uses a truncation or $(\mu+\lambda)$ selection [Sch96] with $\mu = 100$ and $\lambda = 300$, the smart and meta pools use a *rank proportional* selection method, like Teller [TV96]. All plots in this section are based on averages over 30 runs.

Figure 4 shows the progression of the fitness values using the different recombination variants. The figure shows the advantage of *smart random* and *meta self* recombination over *task random* and *smart self* recombination. The results also confirm that self-recombination on the smart level does not work [Tel96] and that self-recombination on the meta level works in our test case. This result indicates that smart operators use a different recombination scheme than meta operators. An interesting result is that meta recombination produces, on aver-

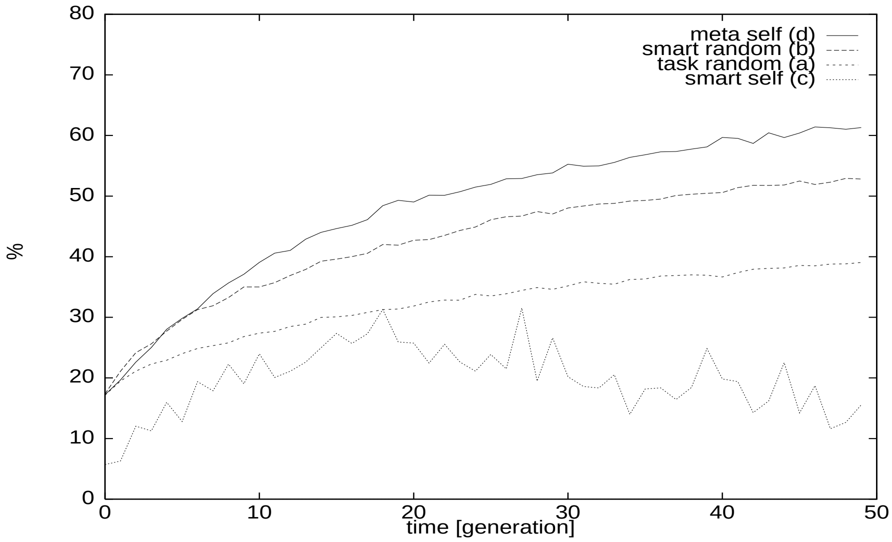

Fig. 4. The curves show the average fitness value of the best 20 individuals of the task level for the four recombination variants in percent of the maximum fitness. Each curve is an average over 30 runs.

age, fitter individuals than smart recombination although the operators use the same operator set.

6.1 Diversity

For measuring the diversity during evolution we define a simple measure of diversity, called *I-O diversity*. The diversity measure is based on the *I-O distance*, which describes the difference of the result values r, between two individuals, for a given set of input data:

Definition 1 (I-O Distance). Let $\mathcal{E}$ be a finite set of input data, then the I-O distance $d(p, p')$ between the individuals $p, p' \in P$ is defined by

$$d(p, p') = \sum_{e \in \mathcal{E}} \begin{cases} 1 & \text{if } r(p, e) \neq r(p', e), \\ 0 & \text{otherwise.} \end{cases} \qquad (4)$$

Definition 2 (I-O Diversity). Let S be a population of individuals out of P and $d : P \times P \rightarrow \mathbb{N}^0$ be the I-O distance. The I-O diversity $\mathcal{D}(S)$ is then defined as

$$\mathcal{D}(S) = \sum_{p \in S} \sum_{p' \in S} d(p, p'). \qquad (5)$$

A high *I-O diversity* value means that the results for a given input are different between different individuals.

Figure 5 indicates that recombination with smart operators uses a different recombination scheme than the random recombination, because the diversity of the random recombination decreases slower than the diversity with smart operators. An interesting aspect of this figure is that the diversity of *meta self* recombination shows the same diversity behaviour as *smart random* and *smart self* recombination although they create in average individuals with poor fitness values.

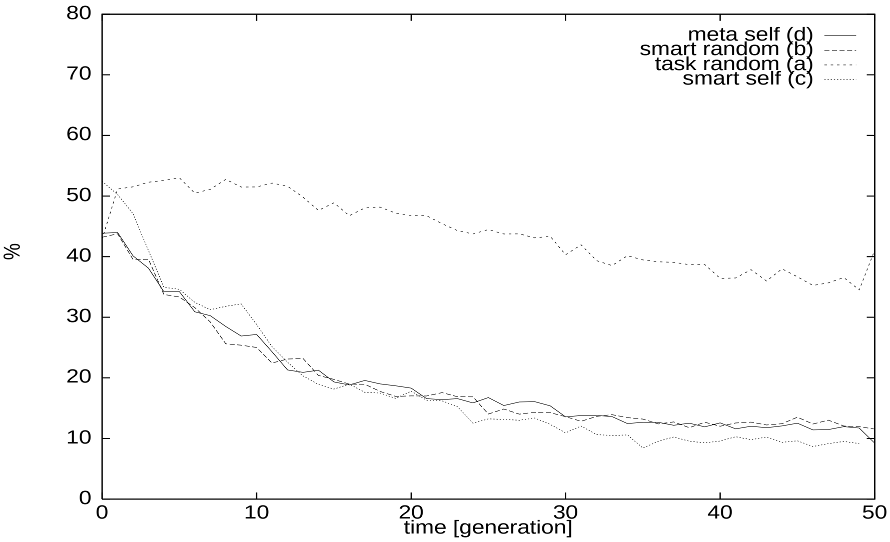

Fig. 5. The I-O diversity in percent of the task level for the different recombination variants.

6.2 Mutation Rate

Another interesting aspect of the smart and meta operators is their ability to select explicitly nodes for mutation by using the *special action function* which marks a node to be mutated. Thus, the mutation rate depends on the smart and meta operators andis therefore subject to evolution and changes over time. The mutation rate of the task level actually defines how often the smart operators use this special operation during the crossover operation in a generation. Figure 6 shows the mutation rate of the task level over the course of generations. The mutation rate for random recombination is 5 % over all generations. The mutation rate for self-recombination is zero during all runs. This mean that no operator uses the possibility to mutate an individual even though it could. The mutation rate in each run starts at about zero. This can be an indication that mutation becomes more important during the evolution. An interesting aspect is that, although smart and meta-recombination result in different mutation rates, the I-O diversity of the individuals is rather the same.

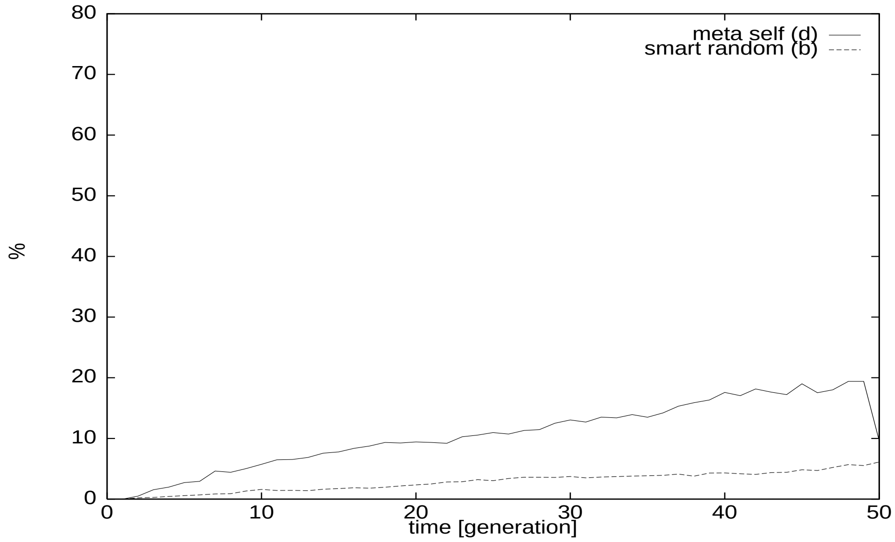

Fig. 6. The average mutation rate of the task level with smart and meta recombination. The mutation rate during self-recombination is zero in all runs.

7 Summary and Outlook

We tested a system which used smart and meta-recombination 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.

The most significant results are that GP programs can be used to perform a crossover operation. In our test case *self* recombination at the level of smart operators does not work well. *Self* recombination at the level of meta operator works. Smart and meta-recombination could find a recombination scheme which is better than random recombination. To say whether these are general phenomena more experiments have to be run on a varity of test cases. We are currently testing meta-evolution different representation on the different level on the SYSGP [BDKB98] system.

Meta-evolution has long been applied in evolutionary algorithms. For example, in evolution strategies (ES) [Sch96] parameters are used which determine the variance and covariance of a generalized n-dimensional normal distribution for mutations. The strategy parameters themselves are adapted during the optimization process. Only the operators used for adaptation are fixed. In our system, the variation of an object on the meta level alters the way how other objects on the meta level are modified, because the object will subsequently interact with others. This "strange loop" and the relation of self-adaption in classical evolutionary algorithms to self-modifying programs should be investigated in the future.

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.

SUPPLEMENT MATERIAL

To ensure reproducibility more detailed information, complete source code, raw experimental data are available from:

http://ls11-www.cs.uni-dortmund.de/gp/meta

References

[BDKB98] M. Brameier, P. Dittrich, W. Kantschik, and W. Banzhaf. SYSGP - A C++ library of different GP variants. Technical Report Internal Report of SFB 531,ISSN 1433-3325, Fachbereich Informatik, Universität Dortmund, 1998.

[BNKF98] W. Banzhaf, P. Nordin, R. Keller, and F. Francone. *Genetic Programming – An Introduction On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, San Francisco und dpunkt.verlag, Heidelberg, 1998.

[Edm98] Bruce Edmonds. Meta-genetic programming: Co-evolving the operators of variation. CPM Report 98-32, Manchester Metropolitan University, 1998.

[FSD97] 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.

[Koz92] J. Koza. *Genetic Programming.* MIT Press, 1992.

[Koz94] J. Koza. *Genetic Programming II.* MIT Press, 1994.

[Nor94] J. P. Nordin. *A Compiling Genetic Programming System that Directly Manipulates the Machinecode.* Cambridge, MIT Press, 1994.

[RS86a] A.E. Rosenberg and F.K. Soong. Evaluation of a vector quantization talker recognition system in text independent and text dependent modes. *Proc. ICASSP*, pages 873– 876, 1986.

[RS86b] A.E. Rosenberg and F.K. Soong. On the use of instantaneous and transitional spectral information in speaker recognition. *Proc. ICASSP*, pages 877– 880, 1986.

[Sch96] H-P. Schwefel. *Evolution and Optimum Seeking.* John Wiley & Sons, Inc., 1996.

[Tel96] 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.

[TV96] A. Teller and M. Veloso. Pado: A new learning architecture for object recognition. In *Symbolic Visual Learning*, pages 81 –116. Oxford University Press, 1996.