Probabilistic Incremental Program Evolution: Stochastic Search Through Program Space

Rafał Sałustowicz and Jürgen Schmidhuber

IDSIA, Corso Elvezia 36, 6900 Lugano, Switzerland
e-mail: {rafal, juergen}@idsia.ch
tel.: +41-91-9919838 fax: +41-91-9919839

Abstract. Probabilistic Incremental Program Evolution (PIPE) is a novel technique for automatic program synthesis. We combine probability vector coding of program instructions [Schmidhuber, 1997], Population-Based Incremental Learning (PBIL) [Baluja and Caruana, 1995] and tree-coding of programs used in variants of Genetic Programming (GP) [Cramer, 1985; Koza, 1992]. PIPE uses a stochastic selection method for successively generating better and better programs according to an adaptive "probabilistic prototype tree". No crossover operator is used. We compare PIPE to Koza's GP variant on a function regression problem and the 6-bit parity problem.

1 Introduction

Probabilistic Incremental Program Evolution (PIPE) synthesizes programs which compute solutions to a given problem. PIPE is inspired by recent work on learning with probabilistic programming languages [Schmidhuber, 1997] and by Population-Based Incremental Learning (PBIL) [Baluja and Caruana, 1995]. PIPE evolves tree-coded programs such as those used in Koza's variant of Genetic Programming [Koza, 1992], from now on simply referred to as GP. For earlier work on Genetic Programming see [Cramer, 1985; Dickmanns, 1987].

PIPE can be applied to any problem that GP can be applied to. PIPE's learning algorithm, however, is more like PBIL's and very different from GP's. PIPE does not use the crossover operator. Instead, it uses a "probabilistic prototype tree" to combine experiences of different programs and to generate better and better programs.

Outline. Section 2 describes the basic data structures and procedures used by PIPE. Section 3 introduces the new learning algorithm. Section 4 compares the performance of PIPE and GP on function regression and 6-bit parity. Section 5 concludes.

2 Basic Data Structures and Procedures

Overview. PIPE generates programs according to an underlying *probabilistic prototype tree.*

Program Instructions. Programs contain instructions from a function set $F = \{f_1, f_2, \ldots, f_k\}$ with k functions and a terminal set $T = \{t_1, t_2, \ldots, t_l\}$ with

l terminals. For instance, to solve a one dimensional function approximation task one might use $F = \{+, -, *, \%, sin, cos, exp, rlog\}$ and $T = \{x, R\}$, where % denotes protected division ($\forall y, z \in \mathbb{R}, z \neq 0$: $y\%z = y/z$ and $y\%0 = 1$), *rlog* denotes protected logarithm ($\forall y \in \mathbb{R}, y \neq 0$: $rlog(y)$=log(abs(y)) and $rlog(0) = 0$), x is an input variable and R represents a generic random constant $\in$ [0;1) (see also "ephemeral random constant" [Koza, 1992]).

Program Representation. Programs are encoded in n-ary trees, with n being the maximal number of function arguments. Each argument is calculated by a subtree. The trees are parsed depth first from left to right. Sample program trees for a function approximation task are shown in Figure 1.

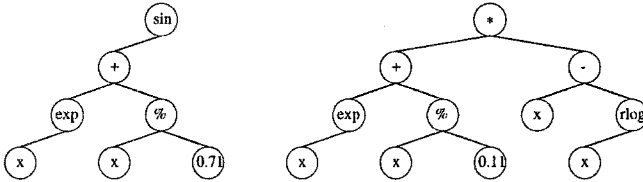


Fig. 1. *Sample program trees for function approximation. Left: f(x)=sin(exp(x)+(x%-0.71)). Right: f(x)=((exp(x)+(x%0.11))*(x-rlog(x)).*

Probabilistic Prototype Tree. The probabilistic prototype tree (*PPT*) is generally a *complete* n-ary tree. At each node $N_{d,w}$ it contains a random constant $R_{d,w}$ and a variable probability vector $\mathbf{P}_{d,w}$, where $d \geq 0$ denotes the node's depth (root node has $d = 0$) and w defines the node's horizontal position when tree nodes with equal depth are read from left to right ($0 \leq w < n^d$). The probability vectors $\mathbf{P}_{d,w}$ have $l + k$ components. Each component $P_{d,w}(I)$ denotes the probability of choosing instruction $I \in F \cup T$ at $N_{d,w}$. We maintain: $\sum_{I \in F \cup T} P_{d,w}(I) = 1$.

Program Generation. To generate a program PROG from *PPT*, an instruction $I \in F \cup T$ is selected with probability $P_{d,w}(I)$ for each accessed node $N_{d,w}$ of *PPT*. This instruction is denoted $I_{d,w}$. Nodes are accessed in a depth first way, starting at the root node $N_{0,0}$, and traversing *PPT* from left to right. Once $I_{d,w} \in F$ is selected, a subtree is created for each argument of $I_{d,w}$. If $I_{d,w} = R$, then an instance of R, called $V_{d,w}(R)$, replaces R in PROG. If $P_{d,w}(R)$ exceeds a threshold T_R, then $V_{d,w}(R) = R_{d,w}$. Otherwise $V_{d,w}(R)$ is randomly generated. Figure 2 illustrates the relation between the prototype tree and a possible program tree. We denote the result of applying PROG to data x PROG(x).

Tree Shaping. To reduce memory requirements we incrementally grow and prune the prototype tree.

Growing. Initially the *PPT* contains only the root node. Nodes are created "on demand" whenever $I_{d,w} \in F$ is selected and the subtree for an argument of $I_{d,w}$ is missing. Figure 3 shows a prototype tree after extraction of two programs.

Pruning. We prune subtrees of the *PPT* attached to nodes which contain at least one probability vector component above a threshold T_P. In case of functions we prune only subtrees that are *not* required as function arguments (see Figure 4). Pruning tends to discard old probability distributions that are irrelevant by now.

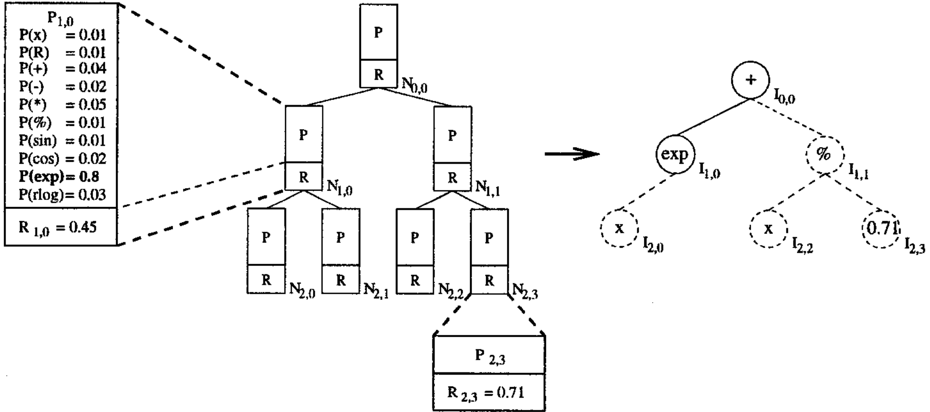


Fig. 2. *Left: example of node $N_{1,0}$'s instruction probability vector* **P** *and random constant R. Middle: probabilistic prototype tree* PPT *with details of node $N_{2,3}$. Right: possible extracted program* PROG. *At the time of creation of instruction $I_{1,0}$ the dashed part of* PROG *did not exist yet. $I_{2,3} = R$ is instantiated to $V_{d,w}(R) = R_{2,3} = 0.71$, because probability $P_{2,3}(R)$ (not shown) exceeds the random constant threshold T_R.*

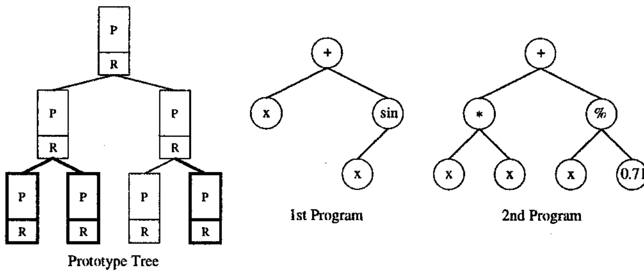


Fig. 3. *Left: prototype tree. Right: two generated programs. The highlighted parts of the prototype tree were created during construction of the second program.*

3 Learning

Overview. PIPE attempts to find better and better programs, program quality being measured by a scalar, real-valued "fitness value". PIPE guides its search to promising search space areas by incrementally building on previous solutions. It generates successive program populations according to the underlying probabilistic prototype tree *PPT* and stores in this tree the knowledge gained from evaluating the programs.

***PPT* Initialization.** Each *PPT* node $N_{d,w}$ requires an initial random constant $R_{d,w}$ and an initial probability $P_{d,w}(I)$ for each instruction $I \in F \cup T$. We pick $R_{d,w}$ uniformly random from the interval [0;1). To initialize instruction probabilities we use a constant probability P_T for selecting an instruction from T and $(1 - P_T)$ for selecting an instruction from F. $\mathbf{P}_{d,w}$ is then initialized as

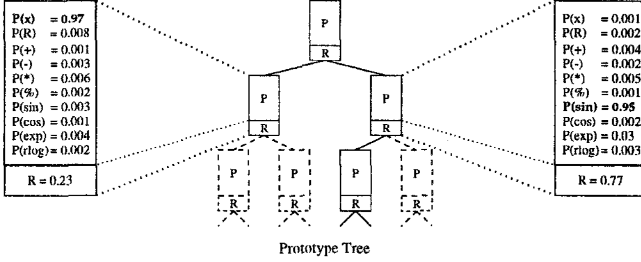


Fig. 4. *The dashed parts of the prototype tree can be pruned, because the probabilities of the adjacent nodes exceed threshold value $T_P = 0.9$ and contain high probabilities for a terminal (left) and a single function with one argument (right).*

follows:

$$P_{d,w}(I) := \frac{P_T}{l}, \forall I : I \in T \qquad \text{and} \qquad P_{d,w}(I) := \tfrac{1-P_T}{k}, \forall I : I \in F \qquad (1)$$

Learning Framework. We combine two forms of learning: Generation-Based Learning (GBL) and Elitist Learning (EL). GBL is PIPE's main learning algorithm. EL's purpose is to make the the best program found so far an attractor. We execute:

1. GBL; 2. REPEAT { with probability P_{el} DO EL otherwise DO GBL }

Here P_{el} is a user-defined constant in [0;1].

Generation-Based Learning. PIPE learns in successive generations, each comprising 5 distinct phases: (1) creation of program population, (2) population evaluation, (3) learning from population, (4) mutation of prototype tree and (5) prototype tree pruning.

(1) Creation of Program Population. A population of programs PROG_j ($0 < j \leq PS$; PS is population size) is generated using the prototype tree PPT as described in Section 2. The PPT is grown "on demand".

(2) Population Evaluation. Each program PROG_j of the current population is evaluated and assigned a non-negative "fitness value" $FIT(\text{PROG}_j)$. If $FIT(\text{PROG}_j) < FIT(\text{PROG}_i)$, then program PROG_j is said to embody a better solution than program PROG_i. Among programs with equal fitness we prefer shorter ones (Occam's razor), as measured by number of nodes. We define b to be the index of the *best* program of the current generation, and preserve the best program found so far in PROG^{el} (elitist).

(3) Learning from Population. Prototype tree probabilities are modified such that the probability $P(\text{PROG}_b)$ of creating PROG_b increases. We call this procedure adapt_PPT_towards(PROG_b). Our experiments indicate that it is beneficial to increase $P(\text{PROG}_b)$ *regardless* of PROG_b's length. To compute $P(\text{PROG}_b)$ we look at all PPT nodes $N_{d,w}$ used to generate PROG_b:

$$P(\text{PROG}_b) = \prod_{d,w:N_{d,w} \text{ used to generate } \text{PROG}_b} P_{d,w}(I_{d,w}(\text{PROG}_b)), \qquad (2)$$

where $I_{d,w}(\text{PROG}_b)$ denotes the instruction of program PROG_b at node position d, w. Then we calculate a target probability P_{TARGET} for PROG_b:

$$P_{TARGET} = P(\text{PROG}_b) + (1 - P(\text{PROG}_b)) \cdot lr \cdot \frac{\varepsilon + FIT(\text{PROG}^{el})}{\varepsilon + FIT(\text{PROG}_b)}. \quad (3)$$

Here lr is a constant learning rate and ε a user defined constant. The fraction $\frac{\varepsilon + FIT(\text{PROG}^{el})}{\varepsilon + FIT(\text{PROG}_b)}$ implements *fitness dependent learning (fdl)*. We learn more from programs with higher quality (lower fitness) than from programs with lower quality (higher fitness). Constant ε determines the degree of *fdl*'s influence. If $\forall$ $FIT(\text{PROG}^{el})$: $\varepsilon << FIT(\text{PROG}^{el})$, then PIPE can use small population sizes, as generations containing only low-quality individuals do not affect the *PPT* much. Even learning with *only one* program per generation is then possible.

Given P_{TARGET}, *all* single node probabilities $P_{d,w}(I_{d,w}(\text{PROG}_b))$ are increased iteratively (in parallel):

REPEAT UNTIL $P(\text{PROG}_b) \geq P_{TARGET}$:
$P_{d,w}(I_{d,w}(\text{PROG}_b)) := P_{d,w}(I_{d,w}(\text{PROG}_b)) + c^{lr} \cdot lr \cdot (1 - P_{d,w}(I_{d,w}(\text{PROG}_b)))$

Here c^{lr} is a constant influencing the number of iterations. We use $c^{lr} = 0.1$, which turned out to be a good compromise between precision and speed.

Finally each random constant in PROG_b is copied to the appropriate node in *PPT*: if $I_{d,w}(\text{PROG}_b) = R$ then $R_{d,w} := V_{d,w}(R)$.

(4) Mutation of Prototype Tree. Mutation is PIPE's major exploration mechanism. Mutation of *PPT* probabilities is guided by the current best solution PROG_b. We want to explore the area "around" PROG_b. Probabilities $P_{d,w}(I)$ stored in all nodes $N_{d,w}$ that were accessed to generate program PROG_b are mutated with a probability P_{M_p}, defined as:

$$P_{M_p} = \frac{P_M}{(l + k) \cdot \sqrt{|\text{PROG}_b|}}, \quad (4)$$

where P_M is a free parameter setting the overall mutation probability and $|\text{PROG}_b|$ denotes the number of nodes in program PROG_b. The justification of the square root is empirical: we found that larger programs improve faster with a higher mutation rate. Selected probability vector components are mutated as follows:

$$P_{d,w}(I) := P_{d,w}(I) + mr \cdot (1 - P_{d,w}(I)), \quad (5)$$

where mr is the mutation rate, another free parameter. All mutated vectors $\mathbf{P}_{d,w}$ are then renormalized.

We see from assignment (5) that small probabilities (close to 0) are subject to stronger mutations than high probabilities. Otherwise mutations would tend to have little effect on the next generation.

(5) Prototype Tree Pruning. At the end of each generation we prune the prototype tree as described in section 2.

Elitist Learning. During elitist learning (EL) we adapt PPT towards the elitist program PROG^{el} by calling adapt_PPT_towards(PROG^{el}), then we prune PPT. However, we neither mutate the probabilities of PPT nor create and evaluate a population, making EL computationally cheap. It focuses search on previously discovered promising parts of the search space. EL is particularly useful with small population sizes. It works efficiently in case of noise-free problems.

Termination Criterion. PIPE is run either for a fixed number of program evaluations PE (time constraint) or until a solution with fitness better than FIT_s is found (quality constraint).

4 Experimental Comparison with GP

In this section we compare our PIPE method to Koza's Genetic Programming variant (GP) on two problems. First, we investigate a continuous function regression problem. We use a non-trivial function to prevent either algorithm from simply guessing it. We then compare both algorithms on the 6-bit parity problem, a discrete task which allows for only 65 distinct fitness values. The limited number of fitness values permits us to test PIPE's built-in Occam's razor.

For both algorithms and problems we set $F = \{+, -, *, \%, sin, cos, exp, rlog\}$ and $T = \{x, R\}$ (see Section 2). For GP R denotes a set of constants from [0;1) ("ephemeral random constant", see [Koza, 1992] for details).

4.1 Function Regression

The function to be approximated is:

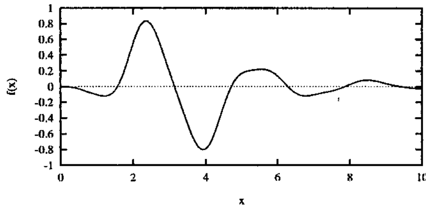
$$f(x) = x^3 \cdot e^{-x} \cdot cos(x) \cdot sin(x) \cdot (sin^2(x) \cdot cos(x) - 1)$$

See Figure 5. The training (testing) data set D_{tr} (D_{te}) samples f at 101 equidistant points from the interval [0;10] ([0.05;10.05]). D_{tr} is used for learning, D_{te} to test generalization. The fitness value for each program PROG is $FIT(\text{PROG}) = \sum_{\forall x \in D_{tr}} |f(x) - \text{PROG}(x)|$, its generalization performance $GEN(\text{PROG}) = \sum_{\forall x \in D_{te}} |f(x) - \text{PROG}(x)|$. We set $PE = 100000$ for both algorithms and tried many parameter settings for both PIPE and GP. Good parameters for PIPE are: P_T=0.8, $\varepsilon = 1$, P_{el}=0.2, PS=10, lr=0.2, P_M=0.2, mr=0.4, T_R=0.3, T_P=0.999999, $FIT_s = 0$. Good parameters for GP are: population size = 2000, crossover rate = 0.9, maximal tree depth = 10, initial depth = 2–6 with "*half and half* population initialization" and "over-selection" – see [Koza, 1992].

Results. 21 independent test runs were conducted for each algorithm. To obtain an idea how generalization performance relates to function approximation quality, consider Figure 6. Note that generalization performance $GEN(\text{PROG}) \approx 20$ can be obtained using a constant function.

PIPE's and GP's generalization performances are summarized in Figure 7. Generalization performances w are plotted against numbers of programs with $GEN(\text{PROG}) \leq w$.

The top 24% of all PIPE runs led to better results than all GP runs. On the other hand, the worst 33% of all PIPE runs led to worse results than all GP runs. PIPE's best solutions are better than GP's, but variance is higher, too.



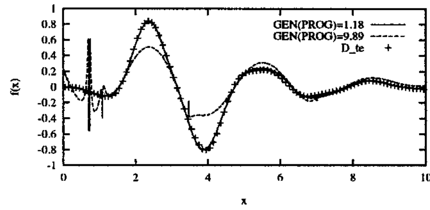
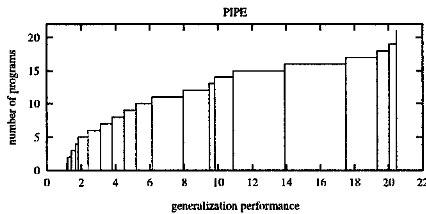


Fig. 5. $f(x) = x^3 \cdot e^{-x} \cdot cos(x) \cdot sin(x) \cdot (sin^2(x) \cdot cos(x) - 1)$

Fig. 6. *Test data set* D_{te} *and approximations with* $GEN(\text{PROG}) = 1.18$ *and* 9.89.



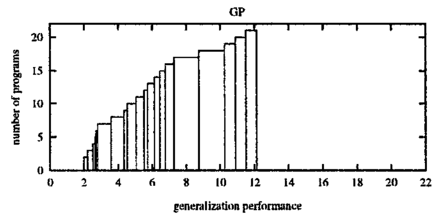


Fig. 7. *Cumulative histograms of PIPE's (left) and GP's (right) generalization performance on the function regression problem. Each box indicates how often PIPE's (left) and GP's (right) generalization performance was at least as good as the corresponding one.*

4.2 6-Bit Parity Problem

For this problem, Boolean values are represented by integers: 1 for true and 0 for false. The 6-bit parity function has 6 Boolean arguments; it returns 1 if the number of non-zero arguments is odd and 0 otherwise.

The fitness of a program is the number of patterns it classifies incorrectly. Best (worst) fitness for classifying all (no) patterns correctly is 0 (64). To fit the Boolean nature of the problem the real-valued output of a program is mapped to 0 if negative and to 1 otherwise. We set $PE = 500000$ for both algorithms. After a coarse parameter search we found the following good parameter settings. For PIPE: P_T=0.6, $\varepsilon = 1$, P_{el}=0.05, PS=12, lr=0.01, P_M=0.4, mr=0.4, T_R=0.3, T_P=0.999999, $FIT_s = 0$. For GP: population size = 2000, crossover rate = 0.9, maximal tree depth = 10, initial depth = 2–6 with "*half and half* population initialization" and "over-selection" – see [Koza, 1992]. The best GP parameters we found turned out to be the same as for the function approximation task, although we tried many combinations. PIPE was less robust with respect to parameter settings.

Results. 50 independent test runs were conducted for each algorithm. The shortest PIPE-program embodying a perfect solution was found after 5829 program evaluations. It has 22 nodes and computes:

```
(x2-((rlog(rlog(cos(0.530687)))%x2)%cos((((x5-x3)-(x0+(x1-x4)))
                    %rlog(0.699001)))))
```

Table 1 summarizes all results.

On this task, PIPE performed better than GP. It solved the problem more reliably (more often) and faster in the median (with fewer program evaluations).

	6-bit parity		
		Program Evaluations	Nodes
Algorithm	solved	min– med –max	min–med–max
PIPE	**70 %**	9,432–**52,476**–482,545	22– **61** –100
GP	60 %	64,000–120,000–396,000	24– 90 –161

Table 1. *Summary of 6-bit parity results.*

PIPE also found less complex solutions (containing fewer nodes).

5 Conclusions

We introduced PIPE, a novel method for automatic program synthesis. Successive generations of programs are generated according to a probabilistic prototype tree *PPT*. The *PPT* guides the search and is updated according to the search results. PIPE performs better than GP on the 6-bit parity problem. It's best solutions to a function regression problem are better than GP's best solutions. GP's results have lower variance though.

References

[Baluja and Caruana, 1995] Baluja, S. and Caruana, R. (1995). Removing the genetics from the standard genetic algorithm. In Prieditis, A. and Russell, S., editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 38–46. Morgan Kaufmann Publishers, San Francisco, CA.

[Cramer, 1985] Cramer, N. L. (1985). A representation for the adaptive generation of simple sequential programs. In Grefenstette, J., editor, *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, Hillsdale NJ. Lawrence Erlbaum Associates.

[Dickmanns et al., 1987] Dickmanns, D., Schmidhuber, J., and Winklhofer, A. (1987). Der genetische Algorithmus: Eine Implementierung in Prolog. Fortgeschrittenenpraktikum, Institut für Informatik, Lehrstuhl Prof. Radig, Technische Universität München.

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

[Schmidhuber, 1997] Schmidhuber, J. (1997). A general method for incremental self-improvement and multi-agent learning in unrestricted environments. In Yao, X., editor, *Evolutionary Computation: Theory and Applications.* Scientific Publ. Co., Singapore. In press.