# H-PIPE: Skip Nodes Facilitate Evolution of Hierarchical Programs

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**

To evolve structured programs we introduce H-PIPE, a hierarchical extension of Probabilistic Incremental Program Evolution (PIPE - Sałustowicz and Schmidhuber, 1997). Structure is induced by "hierarchical instructions" (HIs) limited to top-level, structuring program parts. "Skip nodes" (SNs) inspired by biology's introns (non-coding segments) allow for switching program parts on and off. In our experiments H-PIPE outperforms PIPE, and SNs facilitate synthesis of certain structured programs but not unstructured ones. We conclude that introns can be particularly useful in the presence of structural bias.

**Keywords:** Probabilistic Incremental Program Evolution, Structured Programs, Hierarchical Programs, Introns, Non-Coding Segments.

# 1 Introduction and Previous Work

**Overview.** Hierarchical Probabilistic Incremental Program Evolution (H-PIPE) is a novel method for synthesizing *structured* programs. It uses the PIPE paradigm (Sałustowicz and Schmidhuber, 1997) to iteratively generate successive populations of functional programs from an adaptive probability distribution over all possible programs constructible from a predefined instruction set. As in PIPE the probability distribution is adapted in three ways: (1) Each iteration the probability of the best program in the current population is increased; (2) occasionally the probability of the best program found so far (elitist) is increased; (3) sometimes probabilities are mutated to better explore the search space. H-PIPE uses "hierarchical instructions" (HIs) and "skip nodes" (SNs). HIs can be used to combine lower-level program parts, thus inducing structure. SNs function as gates that allow for keeping program parts dormant without losing them in the course of evolution. In combination with HIs they also enable H-PIPE to substitute program parts by superior partial solutions discovered at later evolutionary stages.

**Structure.** Early genetic programming (GP) work (Dickmanns et al., 1987) as well as Adaptive Levin Search (Schmidhuber, 1997; Schmidhuber et al., 1997b) allow for powerful programs with arbitrary loops etc. Sometimes, however, it is beneficial to introduce inductive bias by appropriately constraining the search space of possible programs. Except for programs evolved by tree-based GP (Cramer, 1985; Koza, 1992), however, not much work has been done on evolution of programs with significant *structural* constraints. There are two such GP variants.

The first reuses program parts, usually in a way less general than that achievable through arbitrary jumps. Typically subprograms are generated and/or extracted from evolved programs; they may then be called in a usually non-recursive fashion from different positions in the code. Examples are: "automatically defined functions" and encapsulation (Koza, 1992), module acquisition (Angeline and Pollack, 1992), adaptive representations through learning (Rosca and Ballard, 1996), automatically defined macros (Spector, 1996). Other approaches do not generate or extract subprograms but restrict GP's recombination operator such that it cannot destroy certain program parts to be reused in the future (e.g., Langdon, 1995; Pringle, 1995; Zannoni and Reynolds, 1997).

The second variant uses grammars to induce structure, constrain the search space, and provide initial bias to speed up evolution. Examples are context-free (Whigham, 1995; Gruau, 1996) or logic grammars (Wong and Leung, 1996).

**Hierarchical Instructions.** H-PIPE's programs are composed of instructions from a fixed instruction set $S = \{I_1, I_2, \ldots, I_z\}$. Each node of the code tree contains an instruction $I$ and can have several son nodes whose instructions are viewed as arguments of $I$. Programs with hierarchical instructions (HIs) are special cases of programs constrained by context-free grammars: We partition $S$ into $m$ disjoint, non-empty instruction sets $S^0, S^1, \ldots, S^m$, and ensure that all "terminal instructions" — instructions with zero arguments — are in $S^0$. Hierarchical order is imposed as follows: Each argument of an instruction in $S^v$ is in $S^v$ or in the "lower level" set $S^{v-1}$. At least one argument must be in $S^{v-1}$, except when $v = 0$. Higher-level instructions can be used to combine program parts made out of lower-level instructions, thus inducing structure.

**Non-Coding Program Parts.** Non-coding program parts ("introns") are those that do not affect the results the program calculates. E.g., in $f(x) = x * 1$, the "$*1$" part is non-coding. Most previous work on non-coding program parts focuses on genetic program synthesis (Blickle and Thiele, 1994; McPhee and Miller, 1995; Nordin et al., 1996; Haynes, 1996; Wineberg and Oppacher, 1996). Usually non-coding program parts evolve or can be inserted to protect coding program parts (parts that *do* affect results calculated by the program) from destructive genetic recombination operators (Blickle and Thiele, 1994; McPhee and Miller, 1995; Nordin et al., 1996; Haynes, 1996). Blickle and Thiele (1994), as well as McPhee and Miller (1995), however, point out that large blocks of non-coding segments in tree-based GP programs cause very slow convergence and difficulties in escaping from local minima. Haynes (1996), on the other hand, shows that artificial removal of non-coding segments from those programs leads to premature convergence. Nordin, Francone, and Banzhaf (1996) investigate the role of non-coding segments in a GP approach based on variable-length strings. They note that non-coding segments may play an important role in finding good solutions and speeding up convergence. Wineberg and Oppacher (1996) use *fixed*-length strings and find that non-coding segments reduce the search space and speed up evolution.

**General observation.** The literature above suggests: in tree-based GP programs with little structure the effect of non-coding segments is twofold. On the one hand they seem necessary to protect blocks of coding segments, on the other hand they can hinder discovery of acceptable solutions. In the case of *structured* programs, however, non-coding program parts can both speed up convergence *and* aid in finding good solutions. Loosely speaking, the more structured the programs (e.g., the greater the restrictions on the coding strings), the higher the potential significance of non-coding segments. Our own experiments with skip nodes will add more empirical evidence in this direction.

**Skip Nodes (SNs).** Much like certain "jump" instructions, skip nodes (SNs) are instructions that allow for skipping program parts. In the context of tree-based functional programs, SNs are functions with $n$ arguments, where $n$ denotes the maximal number of arguments of functions in $S$. SNs return exactly one of their arguments and ignore the others, which thus represent non-coding program parts if $n > 1$. We will demonstrate the benefits of SNs in structuring parts of H-PIPE programs.

**Outline.** Section 2 describes the H-PIPE approach. Section 3 compares the use of HIs and SNs to standard PIPE on function regression and 6-bit parity. Section 4 concludes this paper.

# 2 Hierarchical Probabilistic Incremental Program Evolution

**Overview.** First we will briefly review PIPE (Sałustowicz and Schmidhuber, 1997), which combines probability vector coding of program instructions (Schmidhuber et al., 1997a, 1997b) , Population-Based Incremental Learning (PBIL – Baluja & Caruana, 1995), and tree-coded programs like those used in variants of GP. Then we will describe HIs and SNs in detail.

## 2.1 Probabilistic Incremental Program Evolution (PIPE)

**Overview.** In this section we first summarize PIPE's elementary data structures (programs and the probability distribution). Then we review PIPE's elementary procedures (program generation and "tree shaping"). Finally we list all update rules.

### 2.1.1 Elementary Data Structures

**Program Instructions.** Programs contain instructions from an instruction set $S = \{I_1, I_2, \ldots, I_z\}$ with $z$ instructions. Each instruction $I_j$ $(1 \leq j \leq z)$ is either a function or a terminal. Functions and terminals differ in that the former have one or more arguments and the latter have zero. Thus $S = F \cup T$, where $F = \{f_1, f_2, \ldots, f_k\}$ is a function set with $k$ functions and $T = \{t_1, t_2, \ldots, t_l\}$ is a terminal set with $l$ terminals. Since $F \cap T = \{\}$, $z = k + l$ holds. To solve a one-dimensional function approximation task, e.g., one might use $F = \{+, -, *, \%, sin, cos, exp, rlog\}$ and $T = \{x, R\}$, where % denotes protected division ($\forall y, u \in I\!R, u \neq 0$: $y\%u = y/u$ and $y\%0 = 1$); *rlog* denotes protected logarithm ($\forall y \in I\!R, y \neq 0$: $rlog(y)$=log(abs($y$)) and $rlog(0) = 0$); $x$ is an input variable; and $R$ is a *generic random constant* in [0;1).

**Generic Random Constants.** A generic random constant (GRC) (compare also "ephemeral random constant" (Koza, 1992)) is a zero argument function (a terminal). When accessed during program creation, it is either instantiated to a random value from a predefined, problem-dependent set of constants or a value previously stored together with the probability distribution (see below).

**Program Representation.** Programs are encoded in $n$-ary trees, with $n$ being the maximal number of function arguments. Each nonleaf node encodes a function from $F$ and each leaf node a terminal from $T$. The number of subtrees each node has corresponds to the number of arguments of its function. Each argument is calculated by a subtree. The trees are parsed depth first from left to right.

**Probability Distribution.** The probability distribution is stored in a "probabilistic prototype tree" ($PPT$). The $PPT$ 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 $\vec{P}_{d,w}$, 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 $\vec{P}_{d,w}$ have $z$ components. Each component $P_{d,w}(I)$ denotes the probability of choosing instruction $I \in S$ at $N_{d,w}$. We maintain: $\sum_{I \in S} P_{d,w}(I) = 1$.

***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 $I \in S$. We pick $R_{d,w}$ uniformly random in 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 $F$. $\vec{P}_{d,w}$ is then initialized as follows:

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

### 2.1.2 Elementary Procedures

**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 as $I_{d,w}$. 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 generated uniformly random from a problem-dependent set/interval of constants. Figure 1 illustrates the relation between the prototype tree and a possible program tree.

**Tree Shaping.** A *complete PPT* is infinite. To reduce memory requirements, we incrementally grow and prune the *PPT*.

***Growing.*** Initially, the *PPT* contains only the root node. Further 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.

***Pruning.*** *PPT* subtrees attached to nodes that contain at least one probability vector component above a threshold $T_P$ are pruned. In case of functions, only subtrees that are *not* required as function arguments are pruned.

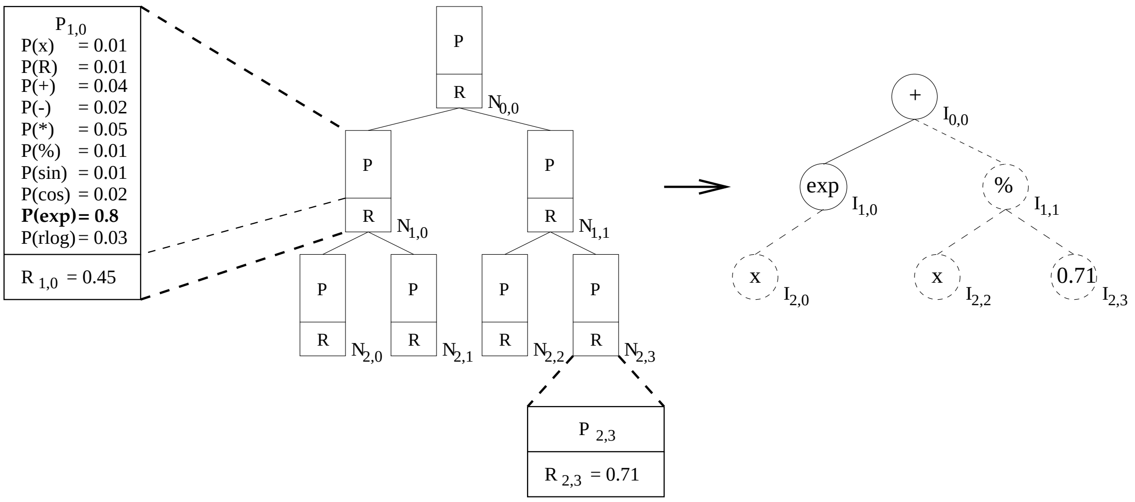


Figure 1: (Left) Example of node $N_{1,0}$'s instruction probability vector $\vec{P}_{1,0}$ and random constant $R_{1,0}$. (Middle) Probabilistic prototype tree $PPT$ with details of node $N_{2,3}$. (Right) Possible extracted program PROG. At the time of creation of instruction $I_{1,0}$, the dashed part of PROG did not yet exist. $I_{2,3} = R$ 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 $T_R$.

#### 2.1.3 Update Rules

**Learning Framework.** PIPE combines two forms of learning: Generation-Based Learning (GBL) and Elitist Learning (EL). It executes:

```
GBL
REPEAT
    with probability P_el DO EL
    otherwise DO GBL
UNTIL termination criterion is reached
```

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

**Generation-Based Learning.** PIPE learns in successive generations, each comprising five 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 $\text{PROG}_j$ $(0 < j \leq PS$; $PS$

is population size) is generated using the prototype tree $PPT$, as described in Section 2.1.2. The $PPT$ is grown "on demand".

***(2) Population Evaluation.*** Each program $\textsc{Prog}_j$ of the current population is evaluated and assigned a scalar, real-valued "fitness value" $FIT(\textsc{Prog}_j)$ reflecting the program's performance on a given task. The "fitness function" mapping programs to fitness values is problem-dependent. PIPE's fitness functions are restricted as follows: (a) Fitness values must not be negative. (b) If $FIT(\textsc{Prog}_j) < FIT(\textsc{Prog}_i)$, then program $\textsc{Prog}_j$ is said to embody a better solution than program $\textsc{Prog}_i$. PIPE seeks to minimize fitness. PIPE also has a built-in Occam's razor: among programs with equal fitness, smaller ones, as measured by number of nodes, are *always* preferred.

***(3) Learning from Population.*** Prototype tree probabilities are modified such that the probability $P(\textsc{Prog}_b)$ of creating $\textsc{Prog}_b$ increases, where $b$ is the index of the *best* program of the current generation. This procedure is called `adapt_PPT_towards`($\textsc{Prog}_b$). First $P(\textsc{Prog}_b)$ is computed by looking at all $PPT$ nodes $N_{d,w}$ used to generate $\textsc{Prog}_b$:

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

where $I_{d,w}(\textsc{Prog}_b)$ denotes the instruction of program $\textsc{Prog}_b$ at node position $d, w$. Then a target probability $P_{TARGET}$ for $\textsc{Prog}_b$ is calculated:

$$P_{TARGET} = P(\textsc{Prog}_b) + (1 - P(\textsc{Prog}_b)) \cdot lr \cdot \frac{\varepsilon + FIT(\textsc{Prog}^{el})}{\varepsilon + FIT(\textsc{Prog}_b)} \tag{3}$$

Here $lr$ is a constant learning rate, $\varepsilon$ a positive user-defined constant, and $FIT(\textsc{Prog}^{el})$ the fitness of $\textsc{Prog}^{el}$, the best program found so far (elitist).

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

REPEAT UNTIL $P(\textsc{Prog}_b) \geq P_{TARGET}$ :

$P_{d,w}(I_{d,w}(\textsc{Prog}_b)) := P_{d,w}(I_{d,w}(\textsc{Prog}_b)) + c^{lr} \cdot lr \cdot (1 - P_{d,w}(I_{d,w}(\textsc{Prog}_b)))$

Here $c^{lr}$ is a constant influencing the number of iterations (usually $c^{lr} = 0.1$). Then, all adapted vectors $\vec{P}_{d,w}$ are renormalized. Finally, each random constant in $\textsc{Prog}_b$ is copied to the appropriate node in *PPT*: if $I_{d,w}(\textsc{Prog}_b) = R$ then $R_{d,w} := V_{d,w}(R)$.

***(4) Mutation of Prototype Tree.*** Probabilities $P_{d,w}(I)$ stored in all nodes $N_{d,w}$ that were accessed to generate program $\textsc{Prog}_b$ are mutated with probability $P_{M_p}$:

$$P_{M_p} = \frac{P_M}{(l+k) \cdot \sqrt{|\textsc{Prog}_b|}} \tag{4}$$

where the user-defined parameter $P_M$ defines the overall mutation probability and $|\textsc{Prog}_b|$ denotes the number of nodes in program $\textsc{Prog}_b$. Selected probability vector components are then mutated as follows:

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

where $mr$ is the mutation rate, another user-defined parameter. All mutated vectors $\vec{P}_{d,w}$ are finally renormalized.

***(5) Prototype Tree Pruning.*** At the end of each generation the prototype tree is pruned, as described in Section 2.1.2.

**Elitist Learning.** During elitist learning (EL), the *PPT* is adapted towards the elitist program $\textsc{Prog}^{el}$ by calling `adapt_PPT_towards(`$\textsc{Prog}^{el}$`)`; then it is pruned.

**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).

## 2.2 Hierarchical Instructions (HIs)

**Overview.** In this section we will first extend PIPE's elementary data structures (programs and probability distribution) to accommodate for HIs. Then we will describe H-PIPE's elementary procedures (program generation and "tree shaping") and update rules.

### 2.2.1 Elementary Data Structures

**Program Instructions.** H-PIPE's programs are composed from $z$ instructions in the instruction set $S = \{I_1, I_2, \ldots, I_z\}$. Each node of the code tree contains an instruction $I$ and can have several son nodes whose instructions are viewed as arguments of $I$. To allow for HIs we partition $S$ into $m$ disjoint, non-empty instruction sets $S^0, S^1, \ldots, S^m$, and ensure that all "terminal instructions" — instructions with zero arguments — are in $S^0$. Hierarchical order arises as follows: Each argument of an instruction in $S^v$ is in $S^v$ or in the "lower level" set $S^{v-1}$. At least one argument must be in $S^{v-1}$, except when $v = 0$. To allow for enforcing descents in the instruction set hierarchy. we add "level down" instructions $\downarrow_i$ to all instruction sets $S^v$, $\forall v : 0 < v \leq m$, where $0 \leq i \leq l(v)$ is the argument index of an instruction $I \in S^v$ with $l(v)$ arguments from $S^{v-1}$. Although "level downs" take a single argument and return it, they are treated as terminal symbols. Thus each instruction set $S^v$ $(0 < v \leq m)$ can be written as $F^v \cup T^v$, where $F^v = \{f_1^v, f_2^v, \ldots, f_{k(v)}^v\}$ is a function set with $k(v)$ functions and $T^v = \{\downarrow_0, \downarrow_1, \ldots, \downarrow_{l(v)}\}$ is a terminal set containing $l(v)$ "level down" instructions. We also have $S^0 = F^0 \cup T^0$, where $F^0 = \{f_1^0, f_2^0, \ldots, f_{k(0)}^0\}$ is a function set with $k(0)$ functions and $T^0 = T$ is a terminal set containing all terminals of $S$ $(l(0) = l)$.

For instance, to structure the function approximation task from Section 2.1 as a linear combination of non-linear parts we split the instruction set $S = \{+, -, *, \%, sin,\ cos, exp, rlog, x, R\}$ into $S^0 = \{*, \%, sin, cos, exp, rlog, x, R\}$ and $S^1 = \{+, -\}$. We then add a $\downarrow_0$ instruction to $S^1$ and obtain $S^1 = \{+, -, \downarrow_0\}$. Function and terminal sets for the lower and upper level then become $F^0 = \{*, \%, sin, cos, exp, rlog\}, T^0 = \{x, R\}$ and $F^1 = \{+, -\}, T^1 = \{\downarrow_0\}$, respectively. Figure 2 shows an example program.

**Program Representation.** With HIs the arity $n(v)$ of a program tree may vary depending on the hierarchical level $v$. On each level $v$, $n(v)$ is the maximal number of function arguments required by functions in $S^v$. For instance, in the function approximation example above, if we add to $S^0$ a three argument function, e.g $**$, where $**(a_1, a_2, a_3) = a_1 * a_2 * a_3$, then the lower-level part of the program tree will be 3-ary while the top-level part will remain 2-ary, as depicted in Figure 3.

**Probability Distribution.** The probability distribution is stored in a "hierarchical probabilistic prototype tree" (*H-PPT*). At each hierarchical level $v (0 \leq v \leq m)$ the *H-PPT* generally contains infinite $n(v)$-ary subtrees $PPT^{dw(v)}$, where the list $dw(v) = ((d_{v+1}, w_{v+1}), (d_{v+2}, w_{v+2}), \ldots,$

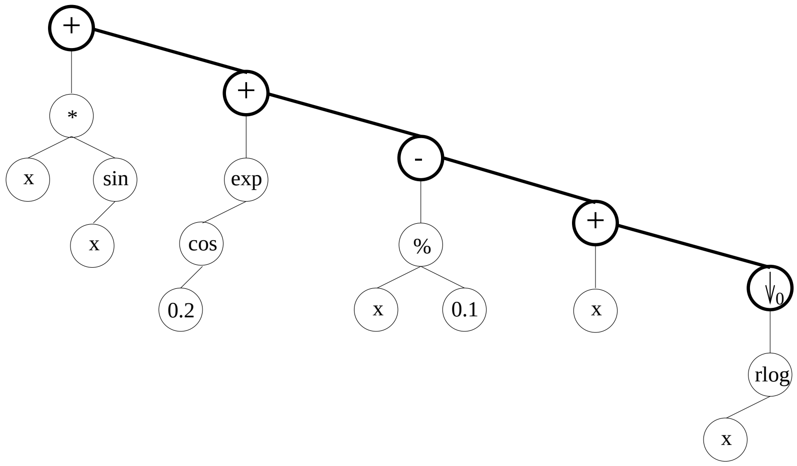


Figure 2: *f(x)=x*sin(x)+exp(cos(0.2))+x%0.1-(x+rlog(x))*. Exemplary program tree for function approximation constrained to a linear combination of non-linear parts. Top-level structuring instructions from $S^1$ appear in boldface.

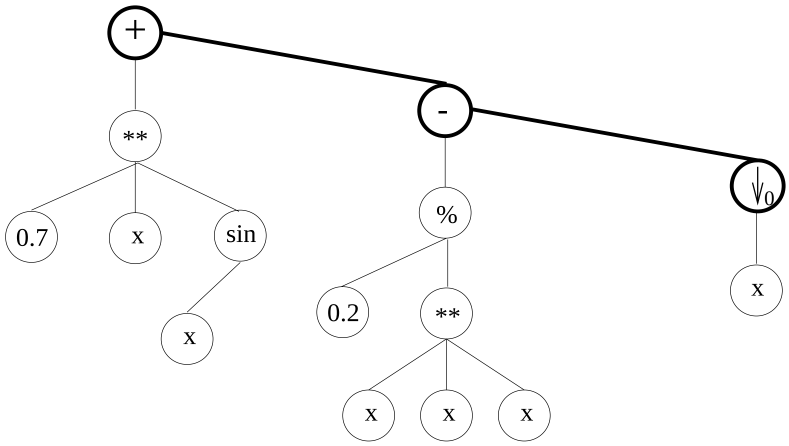


Figure 3: *f(x)=0.7*x*sin(x)+0.2%(x*x*x)-x*. Exemplary program tree for function approximation, with different level-dependent arities. Top-level program parts are 2-ary. Lower level program parts are 3-ary.

$(d_m, w_m))$ describes the absolute position of a subtree: it contains 0 to $m-1$ components depending on the hierarchical position $v$ of $PPT^{dw(v)}$ (0 components, if $v = m$). Each component pair $(d_i, w_i)$ describes the position of a higher level node $N^{dw(i)}_{d_i,w_i}$ in $PPT^{dw(i)}$ to which $PPT^{dw(v)}$ is attached. The position of a node $N^{dw(i)}_{d_i,w_i}$ inside a subtree $PPT^{dw(i)}$ is defined by $N^{dw(i)}_{d_i,w_i}$'s depth $d_i > 0$ ($PPT^{dw(i)}$'s root node has $d_i = 0$) and its horizontal position $w_i$ when subtree nodes with equal depth are read from left to right ($0 \leq w_i < n(i)^{d_i}$). Each node $N^{dw(v)}_{d_v,w_v}$ contains a variable probability vector $\vec{P}^{dw(v)}_{d_v,w_v}$. In addition, each node $N^{dw(0)}_{d_0,w_0}$ contains a random constant $R^{dw(0)}_{d_0,w_0}$. The probability vectors $\vec{P}^{dw(v)}_{d_v,w_v}$, $\forall v : 0 \leq v \leq m$ have $k(v) + l(v)$ components. Each component $P^{dw(v)}_{d_v,w_v}(I)$, $\forall v : 0 \leq v \leq m$ denotes the probability of choosing instruction $I \in S^v$ at $N^{dw(v)}_{d_v,w_v}$. We maintain $\sum_{I \in S^v} P^{dw(v)}_{d_v,w_v}(I) = 1$.

***PPT* Initialization.** Each *H-PPT* node $N^{dw(v)}_{d_v,w_v}$ requires an initial probability $P^{dw(v)}_{d_v,w_v}(I)$ for each instruction $I \in S^v$. Furthermore, each bottom level ($v = 0$) node $N^{dw(0)}_{d_0,w_0}$ requires an initial random constant $R^{dw(0)}_{d_0,w_0}$. We pick $R^{dw(0)}_{d_0,w_0}$ uniformly random in the interval [0;1). To initialize instruction probabilities we use for each hierarchical level $v$ a constant probability $P_{T^v}$ for selecting an instruction from $T^v$ and $(1-P_{T^v})$ for selecting an instruction from $F^v$. $\vec{P}^{dw(v)}_{d_v,w_v}$ is then initialized as follows:

$$P^{dw(v)}_{d_v,w_v}(I) := \frac{P_{T^v}}{l(v)}, \quad \forall I : I \in T^v \qquad \text{and} \qquad P^{dw(v)}_{d_v,w_v}(I) := \tfrac{1-P_{T^v}}{k(v)}, \quad \forall I : I \in F^v \tag{6}$$

### 2.2.2 Elementary Procedures

**Program Generation.** Extracting programs from *H-PPT* is analogous to extracting programs from *PPT*, except that instructions are selected from the appropriate $S^v$, depending on the hierarchical level. To generate a program PROG from *H-PPT*, an instruction $I \in S^v$ is selected with probability $P^{dw(v)}_{d_v,w_v}(I)$ for each accessed node $N^{dw(v)}_{d_v,w_v}$ of *H-PPT*. This instruction is denoted by $I^{dw(v)}_{d_v,w_v}$. Again nodes are accessed in a depth-first way, starting at the root node $N_{0,0}$, and traversing *H-PPT* from left to right. Figure 4 shows a *H-PPT* and a corresponding possible program.

**Tree Shaping.** *H-PPT* shaping is done just like *PPT* shaping (see Section 2.1.2).

### 2.2.3 Update Rules

H-PIPE's update rules are analogous to PIPE's. The only difference is the more sophisticated indexing method due to *H-PPT*'s the hierarchical structure.

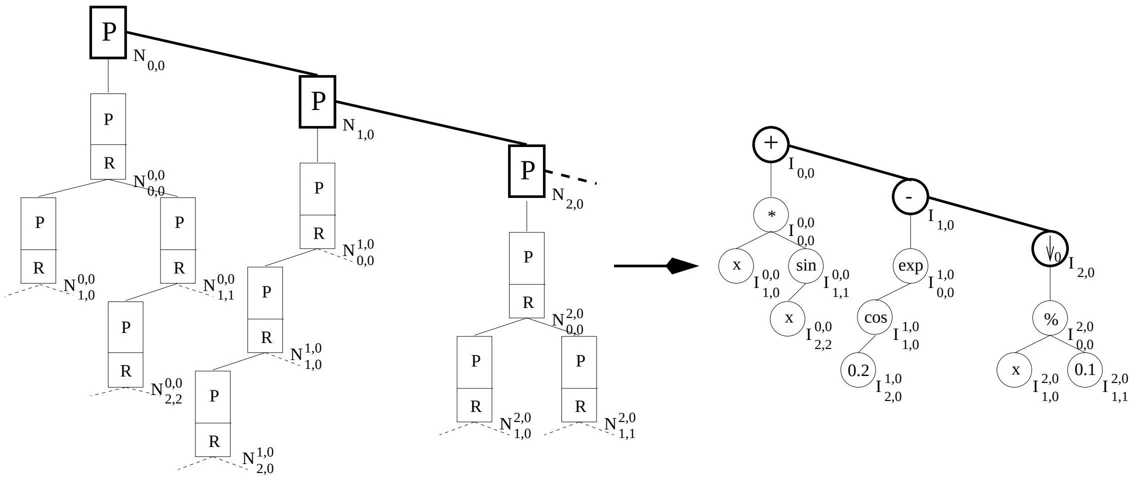


Figure 4: A *H-PPT* (left) and a corresponding possible program (right). The structuring parts of the program are highlighted.

## 2.3 Skip Nodes (SNs)

**Overview.** Skip nodes are inspired by biology's introns. They are functions that serve to switch code parts on and off. We will first define SNs for PIPE, then for H-PIPE. Finally we describe the necessary modifications of PIPE's and H-PIPE's update rules.

### 2.3.1 SNs for PIPE

Let $n$ denote the maximal arity of the PTT (the maximal number of arguments of functions that are not SNs). There are at most $n$ SNs. The $i$-th is denoted $\rightarrow_i$. It is a function with $n$ arguments and returns the $i$-th. Its interpretation is: evaluate the $i$-th argument but ignore the others.

SNs are elements of the function set $F$. For instance, if we add SNs to the instruction set of the function approximation example from Section 2.1.1 we obtain: $F = \{+, -, *, \%, sin, cos, exp, rlog, \rightarrow_0, \rightarrow_1\}$ and $T = \{x, R\}$. Figure 5 shows a PIPE program with SNs. The dashed parts of the program can be viewed as non-coding segments. Note that they need not even be created during program generation and are therefore computationally cheap.

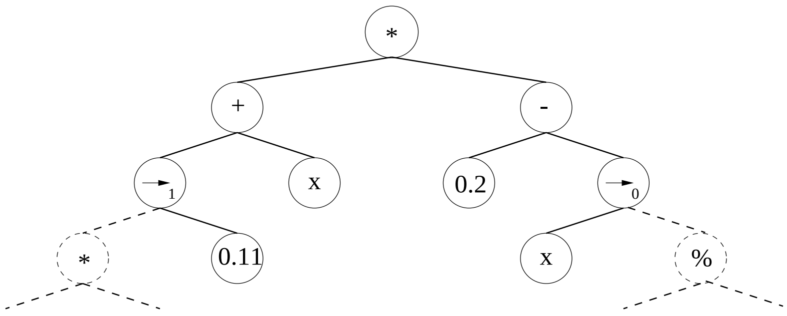


Figure 5: A PIPE program with SNs for function approximation: $f(x) = (0.11 + x) * (0.2 - x)$. The dashed parts of the program are non-coding segments.

#### 2.3.2 SNs for H-PIPE

Let $h(v)$ denote the maximal number of arguments of non-SN functions in $S^v$.

At level $v$ $(0 < v \leq m)$ there are at most $h(v)$ SNs. The $i$-th is denoted $\rightarrow_i^v$. It is a function with $h(v)$ arguments and returns the $i$-th. Its interpretation is: evaluate the $i$-th argument but ignore the others. There are no SNs in $S^0$.

SNs are elements of the function set $F^v$. For instance, if we add SNs to the instruction set of the function approximation example from Section 2.2.1 we obtain: $F^0 = \{*, \%, sin, cos, exp, rlog\}, T^0 = \{x, R\}$ and $F^1 = \{+, -, \rightarrow_0^1\}, T^1 = \{\downarrow_0\}$. Figure 6 shows a H-PIPE program with SNs.

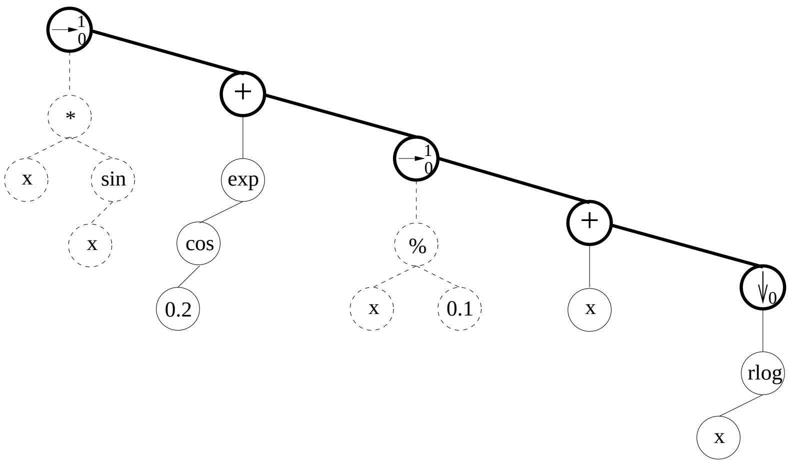


Figure 6: A H-PIPE program with SNs for function approximation: $f(x) = exp(cos(0.2)) + x + rlog(x)$. The dashed parts of the program are non-coding segments.

### 2.3.3 Changes to PIPE's and H-PIPE's Update Rules

**Learning Update Modification.** Parts of *PPT* or *H-PPT* corresponding to non-coding segments are *not* updated.

**Mutation.** In Equation 4 $|\textsc{Prog}_b|$ denotes the number of nodes in $\textsc{Prog}_b$. With SNs, however, $|\textsc{Prog}_b|$ denotes the number of nodes in program $\textsc{Prog}_b$ *without* the non-coding segments created by SNs.

# 3 Experiments

To evaluate the impact of HIs and SNs we cross-compare: (1) PIPE, (2) H-PIPE without SNs (H-PIPE-NO-SN), (3) PIPE with SNs (PIPE-SN), (4) and H-PIPE (PIPE with HIs and SNs in the structuring program parts). To illustrate the significance of *appropriate* initial bias we also test H-PIPE with different structuring instructions (H-PIPE-DIFF). We consider a nontrivial continuous function regression problem and the 6-bit parity problem, a discrete task involving just 65 distinct fitness values. For each combination of learning algorithm and problem we conduct 50-200 independent runs to obtain statistically significant results.

## 3.1 Function Regression

The function to be approximated is

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

which is plotted in Figure 7. The training data set $D_{tr}$ samples $f$ at 101 equidistant points in the interval [0;10]. $D_{tr}$ is used to calculate fitness values during program evolution. Thus, the fitness value of each program $\textsc{Prog}$ is $FIT(\textsc{Prog}) = \sum_{\forall x \in D_{tr}} |f(x) - \textsc{Prog}(x)|$, where $\textsc{Prog}(x)$ denotes the result of applying $\textsc{Prog}$ to data $x$.

**Set-up.** We time-constrain all runs to $PE$ = 100,000 and use the following parameter setting: $P_T$=$P_{T^0}$=$P_{T^1}$=0.8, $\varepsilon$ = 0.000001, $P_{el}$=0.01, $PS$=10, $lr$=0.01, $P_M$=0.4, $mr$=0.4, $T_R$=0.3, $T_P$=0.999999, $FIT_s$ = 0. We use the following instruction sets: (1) PIPE: $F = \{+, -, *,$

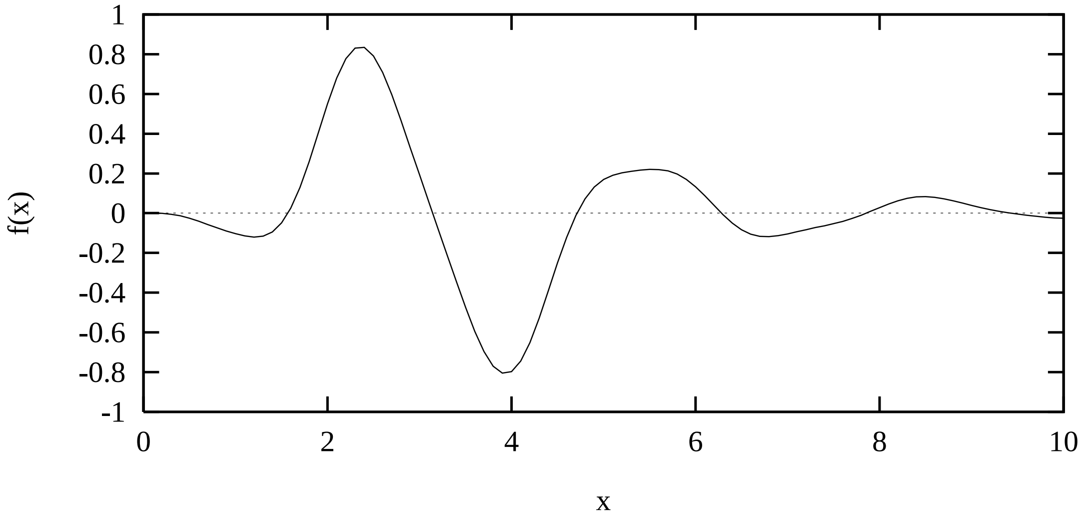


Figure 7: $f(x) = x^3 \cdot e^{-x} \cdot cos(x) \cdot sin(x) \cdot (sin^2(x) \cdot cos(x) - 1)$

$\%, sin, cos, exp, rlog\}$, $T = \{x, R\}$; (2) H-PIPE-NO-SN: $F^1 = \{+, -\}$, $T^1 = \{\downarrow_0\}$, $F^0 = \{*,$ $\%, sin, cos, exp, rlog\}$, $T^0 = \{x, R\}$; (3) PIPE-SN: $F = \{+, -, *, \%, sin, cos, exp, rlog, \rightarrow_0, \rightarrow_1$ $\}$, $T = \{x, R\}$; (4) H-PIPE: $F^1 = \{+, -, \rightarrow_0^1\}$, $T^1 = \{\downarrow_0\}$, $F^0 = \{*, \%, sin, cos, exp, rlog\}$, $T^0 = \{x, R\}$; (5) H-PIPE-DIFF: $F^1 = \{*, \%, \rightarrow_0^1\}$, $T^1 = \{\downarrow_0\}$, $F^0 = \{+, -, sin, cos, exp, rlog\}$, $T^0 = \{x, R\}$.

**Results.** Figure 8 summarizes all results in form of cumulative histograms. We plot performance $u$ against percentage of programs with $FIT(\text{PROG}) \leq u$. Each point indicates the number of programs with $FIT(\text{PROG})$ equal to or better than its x-axis value: algorithms with better performance have more points with smaller x-values.

***PIPE vs. H-PIPE.*** H-PIPE outperforms PIPE. H-PIPE's fitness in the median run is $FIT_{med} = 2.39$, slightly better than PIPE's with $FIT_{med} = 2.55$. In 82% of all runs H-PIPE finds programs with fitness below 4, while only 67% of all PIPE runs accomplish this. On the other hand, the worst 3% of all H-PIPE runs resulted in programs worse than the best found by all PIPE runs. The median of H-PIPE's program size ($Node_{med} = 92$ nodes) is significantly smaller than PIPE's ($Node_{med} = 157$).

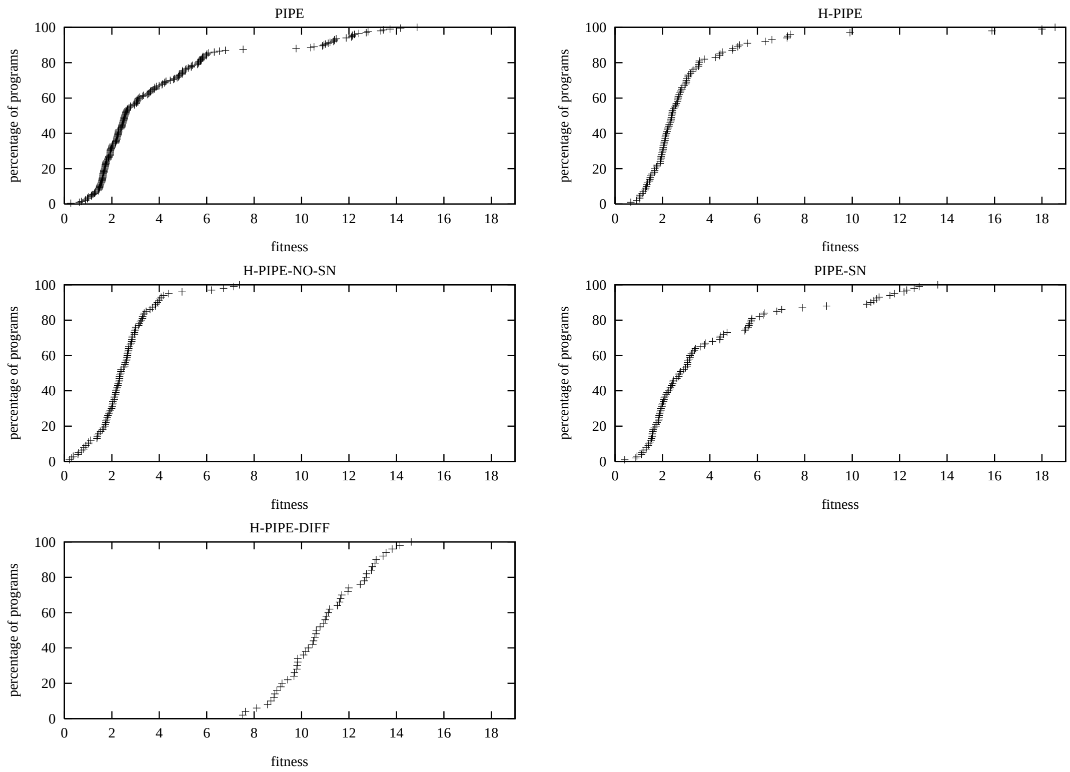


Figure 8: Results for the regression problem.

How much of the performance improvement can be attributed to HIs, how much to SNs? To study this question we now compare PIPE and H-PIPE to PIPE with SNs (PIPE-SN) and H-PIPE without SNs (H-PIPE-NO-SN).

***PIPE and H-PIPE vs. PIPE-SN.*** PIPE-SN performs much like PIPE, and worse than H-PIPE. PIPE-SN's $FIT_{med} = 2.70$ is slightly higher than PIPE's ($FIT_{med} = 2.55$). Like PIPE, in 67% of all runs PIPE-SN found programs with fitness below 4. Its worst programs are slightly better than the worst program among the best of the individual PIPE runs. PIPE-SN's programs ($Node_{med} = 117$) tend to be smaller than PIPE's ($Node_{med} = 157$), but larger than H-PIPE's ($Node_{med} = 92$).

We observe that SNs in unstructured PIPE programs are neither harmful nor beneficial.

***PIPE and H-PIPE vs. H-PIPE-NO-SN.*** H-PIPE-NO-SN is the best competitor, slightly better than H-PIPE, much better than PIPE. H-PIPE-NO-SN's $FIT_{med} = 2.38$ is roughly as good as H-PIPE's $FIT_{med} = 2.39$. In 91% of all runs , however, H-PIPE-NO-SN found programs with fitness below 4, compared to H-PIPE's 82% and PIPE's 67%. Furthermore, unlike with H-PIPE and PIPE, no program found by H-PIPE-NO-SN has fitness above 7.39. The median size of H-PIPE-NO-SN programs, $Node_{med} = 96$, is roughly the same as H-PIPE's ($Node_{med} = 92$) and significantly smaller than PIPE's ($Node_{med} = 157$).

For this particular problem we observe that HIs by themselves increase PIPE's performance more than SNs. Later (in Section 3.2) we will see, however, that indeed both HIs *and* SNs are needed to solve certain tasks more efficiently. But first we will illustrate the importance of choosing the right HIs.

***PIPE and H-PIPE vs. H-PIPE-DIFF.*** H-PIPE-DIFF performs significantly worse than H-PIPE and PIPE. The fitness of the best program found by H-PIPE-DIFF in 50 independent runs is only 7.52. H-PIPE-DIFF's median fitness $FIT_{med} = 10.62$. Compare H-PIPE's and PIPE's, which are 2.39 and 2.55, respectively.

This demonstrates, not unexpectedly, that appropriate initial bias due to "good" HIs is crucial to H-PIPE's success.

**Conclusion.** HIs can increase PIPE's performance significantly. They need to be selected carefully, however. SNs do not contribute much to solving the function regression task. In case of PIPE they reduce program size without affecting solution quality. In case of H-PIPE they have a slightly detrimental effect on overall performance.

The next experiment, however, will show that for some tasks only the combination of HIs *and* SNs leads to significant performance improvement.

### 3.2 6-Bit Parity

The 6-bit parity function has six Boolean arguments represented by integers: 1 for true and 0 for false. It returns 1 if the number of nonzero 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). We use all 64 patterns for training.

**Set-up.** We time-constrain all runs to $PE$ = 500,000 and use the following parameter settings: $P_T$=$P_{T^0}$=$P_{T^1}$=0.6, $\varepsilon$ = 0.000001, $P_{el}$=0.01, $PS$=10, $lr$=0.01, $P_M$=0.4, $mr$=0.4, $T_R$=0.3, $T_P$=0.999999, $FIT_s = 0$. Note that, except for $P_T$, $P_{T^0}$, and $P_{T^1}$, all parameters are set to the same values as for the function regression task (see Section 3.1). Most of PIPE's and H-PIPE's parameters seem robust with respect to changing tasks. We use the following instruction sets: (1) PIPE: $F = \{+, -, *, \%, sin, cos, exp, rlog\}$, $T = \{x_0, x_1, x_2, x_3, x_4, x_5, R\}$; (2) H-PIPE-NO-SN: $F^1 = \{*, \%\}$, $T^1 = \{\downarrow_0\}$, $F^0 = \{+, -, sin, cos, exp, rlog\}$, $T^0 = \{x_0, x_1, x_2, x_3, x_4, x_5, R\}$; (3) PIPE-SN: $F = \{+, -, *, \%, sin, cos, exp, rlog, \rightarrow_0, \rightarrow_1\}$, $T = \{x_0, x_1, x_2, x_3, x_4, x_5, R\}$; (4) H-PIPE: $F^1 = \{*, \%, \rightarrow_0^1\}$, $T^1 = \{\downarrow_0\}$, $F^0 = \{+, -, sin, cos, exp, rlog\}$, $T^0 = \{x_0, x_1, x_2, x_3, x_4, x_5, R\}$; (5) H-PIPE-DIFF: $F^1 = \{+, -, \rightarrow_0^1\}$, $T^1 = \{\downarrow_0\}$, $F^0 = \{*, \%, sin, cos, exp, rlog\}$, $T^0 = \{x_0, x_1, x_2, x_3, x_4, x_5, R\}$. 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.

**Results.** Table 1 summarizes all results. The first column displays for each algorithm the

Table 1: Summary of 6-bit parity results. Best values are in boldface.

| | 6-bit parity | | |
|---|---|---|---|
| Algorithm | solved | Program Evaluations min– med –max | Nodes min–med–max |
| H-PIPE | **94** % | 5,700–**37,460**–397,000 | 23– 61 –96 |
| PIPE | 79 % | 3,520– 79,950 –497,220 | 24– 64 –137 |
| PIPE-SN | 76 % | 1,676– 73,720 –487,930 | 25– 58 –110 |
| H-PIPE-NO-SN | 66 % | 3,720–166,740–468,950 | 21– **49** –85 |
| H-PIPE-DIFF | 28 % | 38,300–216,570–457,330 | 24– 61 –94 |

percentage of independent runs leading to perfect solutions within the given time frame ($PE$). The next three columns show the numbers of program evaluations necessary to find perfect solutions in the shortest, median, and longest run, respectively. The final three columns list the minimal, median, and maximal program sizes embodying perfect solutions.

**Comparison.** H-PIPE performs best. It solves the task more often and significantly faster (with less program evaluations) than PIPE, PIPE with SNs, and H-PIPE without SNs. PIPE and PIPE-SN have roughly the same performance. PIPE-SN finds slightly fewer solutions, but is faster than PIPE in the median run. The median size of its solutions is also slightly smaller than PIPE's. Although its solution size is smallest in the median run, H-PIPE-NO-SN performs significantly worse than PIPE and PIPE-SN. It finds fewer solutions and requires more than twice as many program evaluations (in the median run). H-PIPE-DIFF with wrong initial bias is worst of all. It needs more than five times as many program evaluations as H-PIPE to find roughly three times fewer solutions.

**Conclusion.** With this particular task H-PIPE outperforms PIPE. Neither SNs by themselves nor HIs by themselves are able to improve PIPE's performance. In absence of structure SNs' effects are neither harmful nor beneficial, while HIs by themselves decrease PIPE's performance. The combination of both HIs (embodying the proper initial bias) and SNs in H-PIPE, however, allows for significant improvement.

# 4 Conclusion

H-PIPE, a novel method for synthesizing *structured* programs, uses hierarchical instructions (HIs) to structure programs and skip nodes (SNs) to facilitate their synthesis. HIs combine program parts, while SNs allow for introns (non-coding segments). In our experiments, SNs by themselves were useless for improving performance. Sometimes HIs by themselves worked extremely well, but not always. Then, however, SNs were crucial to achieve dramatic improvement.

Our review of previous work on non-coding segments suggests that non-coding segments seem to require *structured* code to unfold their benefits. Our own results add further empirical evidence in this vein.

# Acknowledgments

Thanks to Marco Wiering, Nicol Schraudolph, and Jieyu Zhao for valuable comments and suggestions that helped to improve a draft of this paper.

# References


Angeline, P. J. and Pollack, J. B. (1992). The evolutionary induction of subroutines. In *Proceedings of the 14th Annual Conference of the Cognitive Science Society*, pages 236–241, Hillsdale, NJ. Lawrence Erlbaum Associates.

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.

Blickle, T. and Thiele, L. (1994). Genetic programming and redundancy. In Hopf, J., editor, *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*, pages 33–38, Im Stadtwald, Building 44, D-66123 Saarbrücken, Germany. Max-Planck-Institut für Informatik (MPI-I-94-241).

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*, pages 183–187, Hillsdale, NJ. Lawrence Erlbaum Associates.

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.

Gruau, F. (1996). On using syntactic constraints with genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 19, pages 377–394. MIT Press, Cambridge, MA, USA.

Haynes, T. (1996). Duplication of coding segments in genetic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 344–349, Portland, OR.

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

Langdon, W. B. (1995). Directed crossover within genetic programming. Research Note RN/95/71, University College London, Gower Street, London WC1E 6BT, UK.

McPhee, N. F. and Miller, J. D. (1995). Accurate replication in genetic programming. In Eshelman, L., editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 303–309, Pittsburgh, PA, USA. Morgan Kaufmann.

Nordin, P., Francone, F., and Banzhaf, W. (1996). Explicitly defined introns and destructive crossover in genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 6, pages 111–134. MIT Press, Cambridge, MA, USA.

Pringle, W. R. (1995). ESP: Evolutionary structured programming. Technical report, Penn State University, Great Valley Campus, PA, USA.

Rosca, J. P. and Ballard, D. H. (1996). Discovery of subroutines in genetic programming. In Angeline, P. and K. E. Kinnear, J., editors, *Advances in Genetic Programming 2*, page Chapter 9. MIT Press, Cambridge, MA.

Sałustowicz, R. P. and Schmidhuber, J. (1997). Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141.

Schmidhuber, J. (1997). Discovering neural nets with low Kolmogorov complexity and high generalization capability. *Neural Networks*, 10(5):857–873.

Schmidhuber, J., Zhao, J., and Schraudolph, N. (1997a). Reinforcement learning with self-modifying policies. In Thrun, S. and Pratt, L., editors, *Learning to learn*. Kluwer. in press.

Schmidhuber, J., Zhao, J., and Wiering, M. (1997b). Shifting inductive bias with success-story algorithm, adaptive Levin search, and incremental self-improvement. *Machine Learning*, 28:105–130.

Spector, L. (1996). Simultaneous evolution of programs and their control structures. In Angeline, P. and K. E. Kinnear, J., editors, *Advances in Genetic Programming 2*, page Chapter 7. MIT Press, Cambridge, MA, USA.

Whigham, P. A. (1995). Grammatically-based genetic programming. In Rosca, J. P., editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, Tahoe City, California, USA.

Wineberg, M. and Oppacher, F. (1996). The benefits of computing with introns. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 410–415, Stanford University, CA, USA. MIT Press.

Wong, M. L. and Leung, K. S. (1996). Evolving recursive functions for the even-parity problem using genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 11, pages 221–240. MIT Press, Cambridge, MA, USA.

Zannoni, E. and Reynolds, R. G. (1997). Learning to control the program evolution process with cultural algorithms. *Evolutionary Computation*, 5(2):181–211.