Exact Uniform Initialization For Genetic Programming

Walter Böhm
Department of Mathematical Methods of Statistics,
Institute of Statistics,
Vienna University of Economics and Business
Administration, Augasse 2-6, A-1090 Vienna, Austria.
E-mail: `boehm@wu-wien.ac.at`

Andreas Geyer-Schulz
Department of Applied Computer Science,
Institute of Information Processing
Vienna University of Economics and Business
Administration, Augasse 2-6, A-1090 Vienna, Austria.
E-mail: `geyers@wu-wien.ac.at`

Abstract

In this paper we solve the problem of *exactly uniform* generation of complete derivation trees from k-bounded context-free languages. The result is applied and is used for developing an exact uniform initialization routine for a genetic programming variant based on an explicit representation of the grammar of the context-free language (simple genetic algorithm over k-bounded context-free languages) [Geyer-Schulz, 1996b]. In this genetic programming variant the grammar is used to generate complete derivation trees which constitute the genomes for the algorithm. For the case that no a priori information about the solution is available, we prove that this (simple random sampling) algorithm is optimal in the sense of a minimax strategy. An exact uniform initialization routine for Koza's genetic programming variant [Koza, 1992] is derived as a special case.

1 INTRODUCTION

The random generation of combinatorial structures is of major theoretical and practical importance for the design of randomized algorithms [Gupta *et al.*, 1994] and often leads to

difficult and unsolved combinatorial problems. In genetic programming the initialization of a population of programs is usually tackled by ad hoc methods. A few examples from the literature are:

1. Koza's "full method" (naive), Koza's "grow method" (naive) and Koza's "ramped-half-and-half" method [Koza, 1992, p. 93 and 597f]. Koza's "full method" is a recursive tree generation method which generates "full" parse trees whose leaves all have paths of the same length to the root. Koza's "grow" method generates parse trees whose leaves have paths of different length to the root. Koza's "ramped-half-and-half" method uses the "full" method for generating one half of the members of the population and the "grow" method for the other half with the maximum depth varying between 2 and a fixed upper bound.
2. A "naive" method with and without duplicates [Geyer-Schulz, 1995] and [Whigham, 1995a]. The "naive" method of generating a derivation tree from a grammar consists of expanding the nonterminal symbols of the frontier of a derivation tree with the symbols on the right hand side of a production rule for this nonterminal symbol until only terminal symbols are contained in the frontier. If more than one production rule is available for a nonterminal symbol, the production is chosen with equal probability. The process starts from the nonterminal symbol specified as the start symbol of the grammar and is usually terminated after some maximum number of "expansions" or derivation steps.
3. "Stratified sampling" with a rejection method [Geyer-Schulz, 1995] and [Iba, 1995], [Iba, 1996a] and [Iba, 1996b]. Trees are generated by any method and only trees satisfying a sampling plan are accepted as members of the population. The sampling plan specifies how many trees with e.g. a certain number of nodes or requiring a certain number of derivation steps for construction have to be in the population.
4. "Compound derivations" [Geyer-Schulz, 1996a] and [Geyer-Schulz, 1996b], or "grammar bias" [Whigham, 1995b], [Whigham, 1996]). The basic idea of this method is to find a grammar which generates the same language as the original grammar, but which favors the construction of larger parts of a derivation tree in a single derivation step. This effect can be achieved by simply adding redundant production rules. For example, in the grammar shown in Figure 1 the production rule <fe> := "(" <f2> <fe> <fe> ")" ; may be added a second time.

```
S := <fe> ;
<fe> := "(" <f0> ")" |
        "(" <f1> <fe> ")" |
        "(" <f2> <fe> <fe> ")"    ;
<f0> := "D1" | "D2" ;
<f1> := "NOT" ;
<f2> := "OR" | "AND" ;
```

Figure 1: The Backus Naur Form of L_{XOR}

The genetic programming variant used in this paper, simple genetic algorithms over k-bounded context-free languages, is characterized by an explicit representation of the grammar of the context-free language and by a bound k on the number of derivation steps

available for generating a derivation tree [Geyer-Schulz, 1996b]. Instead of bit strings as in the canonical genetic algorithm or parse trees as in Koza's genetic programming variant the algorithm works with complete derivation trees. For the definitions and results about context-free languages which are needed in this paper we refer the reader to section 2.

Figure 1 shows the grammar of a context-free language for solving the XOR problem in Backus Naur Form. For example, let us compute the probability of generating the program (NOT(D2)) from the grammar shown in Figure 1 by the "naive" method of [Geyer-Schulz, 1995] and [Whigham, 1995a]: We start by expanding the start symbol <fe>. For this substitution we choose either "(" <f0> ")" or "(" <f1> <fe> ")" or "(" <f2> <fe> <fe> ")". The probability of making the right choice is 1/3. Suppose, we picked "(" <f1> <fe> ")" and we continue by expanding <f1>. In this derivation step we have no choice. With probability 1 we get "NOT". From "(" "NOT" <fe> ")" we expand <fe> in the third derivation step. The probability of choosing "(" <f0> ")" is 1/3. From "(" "NOT" "(" <f0> ")" ")" we expand <f0> and we can choose from either "D1" or "D2". In the fourth derivation step we get (NOT(D2)) with probability 1/2. To compute the probability of deriving (NOT(D2)) we multiply the probabilities of choosing the right production in each derivation step: $1/3 \times 1 \times 1/3 \times 1/2 = 1/18 = 0.05556$. In Table 1 we present a few examples of the probabilities of generating programs from the grammar shown in Figure 1 for an increasing number of derivation steps. (The examples shown have been selected arbitrarily.) In Table 1 we observe an exponential decay in the probability of generating long programs, when choosing production rules in Figure 1 with equal probability.

Table 1: Programs and the Probability of Derivation (n = Number of Derivation Steps Required)

Program	P(Program)	n
(D1)	0.16667	2
(NOT(D2))	0.05556	4
(NOT(NOT(D2)))	0.01851	6
(OR(AND(D2)(NOT(D2)))(D2))	0.00004	12
(AND(D2)(NOT(AND(NOT(D2))(NOT(D1)))))	$4.76 \cdot 10^{-6}$	16
(NOT(NOT(NOT(NOT(NOT(NOT (AND(D1)(D1))))))))	$6.35 \cdot 10^{-6}$	18
(AND(OR(NOT(NOT(D1))) (AND(D1)(D2)))(NOT(D2)))	$1.32 \cdot 10^{-7}$	20
(OR(D1)(NOT(AND(AND(D2) (NOT(NOT(OR(D2)(D2))))) (NOT(OR(D1)(D2))))))	$3.40 \cdot 10^{-11}$	30

These methods have the disadvantage that the population of programs ceases to be a simple random sample, where each population of n programs has the same chance to be selected. In the "naive" methods the skewed distribution of programs leads to a degraded genetic programming performance. Each of the authors quoted above has intuitively recognized this fact and tried to get a "better" mixture of programs by ad hoc methods. All authors report that these ad hoc methods resulted in improved genetic programming performance. However, the "best" mixture of programs is obtained by simple random sampling and intuitively this implies that – without additional information – simple random sampling is the best algorithm for initializing a population of programs.

In this paper we solve the problem of *exactly uniform* generation of complete derivation trees from k-bounded context-free languages. The resulting algorithm has all properties of simple random sampling. The tree generating algorithm is based on a k-bounded, recursive derivation tree generator and a nonlinear transformation on the probability of choosing the next production rule. The nonlinear transformation is based on the automatic derivation of a "word" counting function from the Backus Naur Form (BNF) of the (unambiguous) grammar of a context-free language from [Geyer-Schulz, 1996b]. For ambiguous grammars we count "complete derivation trees". We count distinct trees, which means different derivations of the same word are considered distinct. In Section 2 we introduce the notation, definitions and results for context-free languages. An example of "counting" is given in Section 3. The general technique is presented in Section 4 and its implementation in APL is deferred to the Appendix. In Section 5 the results of the previous sections are used to implement an exact, uniform initialization algorithm for k-bounded context-free languages. In Section 6 we show that the algorithm behaves as expected and in Section 7 we discuss the impact on genetic programming performance. For the case that no a priori information about the solution is available, we prove that this (simple random sampling) algorithm is optimal in the sense of a minimax strategy. Finally, in Section 8 an exact uniform initialization routine for Koza's genetic programming variant [Koza, 1992] is derived as a special case.

2 CONTEXT-FREE LANGUAGES

In this section we introduce the notation, definitions and results of context-free languages which are used in the rest of the paper.

By $L(G)$ we mean the language L generated by grammar G, this is the set of sentences (words) generated by G. By a *k-bounded* language $L(G)$ we mean the set of words generated by G with at most k derivation steps.

A *context-free grammar* G is a 4-tuple $G = (V_{NT}, V_T, P, S)$, where V_{NT} is a finite set of nonterminal symbols, V_T is a finite set of terminal symbols disjoint from V_{NT}, P is a finite subset of $V_{NT} \times (V_{NT} \cup V_T)^*$ called the production rules or productions of the grammar and S is a distinguished symbol in V_N called the start symbol of G [Aho and Ullman, 1972]. We denote the empty word by ϵ.

A *sentential form* of G is defined recursively: S is a sentential form and if xyz is a sentential form and $y \rightarrow u$ is in P, then xuz is a sentential form too.

A *sentence* or a *word* w of $L(G)$ is a sentential form without terminal symbols. Clearly, the *programs* in genetic programming are words or sentences of a context-free language.

For specifying small grammars a very compact notation is used (e.g. [Aho and Ullman, 1972]): Symbols from V_T are taken from the small letters $a, \ldots, z$, symbols from V_{NT} are taken from the capital letters $A, \ldots, Z$ and *derives* is denoted by $\rightarrow$. For the set of productions $P = \{S \rightarrow AB, S \rightarrow aS\}$ we use the production $S \rightarrow AB \mid aS$ as shorthand. $\mid$ denotes *or*.

For "real" grammars we use a version of Backus Naur Form (BNF) [Naur, 1963]: Symbols from V_T are delimited by ' ', for example ' 'D1' ', symbols from V_{NT} are delimted by < and >, for example <fe>. *Derives* is denoted by :=, *or* by |, and *catenation* is denoted by juxtaposition of symbols. Whenever we want to represent *catenation* explicitly, we denote *catenation* as $\star$, for example aS corresponds to $a \star S$ with explicit representation of the

catenation operation. For simplifying automatic processing of grammars in Backus-Naur form we use a semicolon ; as production rule separator and we include the specification of the start symbol as the *first* derivation. For example, S: = <fe> ; as first production specifies that <fe> is the start symbol of the grammar in Figure 1.

Both notations are isomorphic. However, in practice Backus Naur Form is more convenient for specifying large languages manually. By delimiting symbols no confusion about symbols and symbol strings can arise as for example in [Whigham, 1995b]. It is common practice to use informal names for syntactic categories to make grammars more readable. Figure 1 shows a grammar of a context-free language for solving the XOR problem in Backus Naur Form. For the syntactic categories we have used the following informal memnonic scheme: <fe> is short for functional expression, <f0> denotes variables (functions with 0 arguments), <f1> denotes functions with 1 argument, and <f2> denotes functions with 2 arguments.

A *derivation tree* D is a labeled ordered tree for a context-free grammar $G = (V_N, V_T, P, X)$ with the following properties: X labels the root of D. For all subtrees $D_1, \ldots, D_k$ of the sons of the root X with the root of the subtree D_i labeled X_i, $X \rightarrow X_1 \ldots X_k$ is a production of P. If X_i is a nonterminal symbol, D_i is a derivation tree, if X_i is a terminal symbol, D_i is the single node X_i. If the empty word ϵ is the root of D_1, the only subtree of D, then $X \rightarrow \epsilon$ is a production in P [Aho and Ullman, 1972, p. 139].

The *frontier* of a derivation tree is the string obtained by concatenating the leaves of the derivation tree (in order from the left) [Aho and Ullman, 1972, p. 140].

$\Longrightarrow$ denotes the relation *derives*, $\overset{k}{\Longrightarrow}$ denotes the k-fold product of the relation $\Longrightarrow$ and $\overset{*}{\Longrightarrow}$ denotes the k-fold product of the relation $\Longrightarrow$ for an arbitrary but finite k.

Theorem 2.1 *Suppose $G = (V_{NT}, V_T, P, X)$ is a context-free grammar. Then $X \overset{*}{\Longrightarrow} \alpha$ if and only if there is a derivation tree with the sentential form α as frontier [Aho and Ullman, 1972, p. 143].*

Proof See [Aho and Ullman, 1972, p. 141] ∎

From theorem 2.1 the following consequences are obvious: For each word w of $L(G)$ there exists at least one derivation tree with frontier w. We can retrieve a word w from its derivation tree by extracting the frontier of the derivation tree.

A *complete derivation tree* is a derivation tree whose frontier is a word w of $L(G)$. All leaves of a complete derivation tree are terminal symbols and all interior nodes of a complete derivation tree are nonterminal symbols.

The leftmost (rightmost) derivation associated with a derivation tree is unique. If there exists at least one word in $L(G)$ which has two or more distinct leftmost (rightmost) derivations, we say that the grammar G is *ambiguous*. A language which has no unambiguous grammar is called *inherently ambiguous*. The counting algorithm presented in section 4 counts all complete derivation trees once. However, the implication that every word is counted once only holds for unambiguous grammars. How often a word will be counted for ambiguous grammars, depends on the degree of ambiguity of the grammar which may be infinite [Kuich and Salomaa, 1986, p. 296]. Consider for example the grammar G with the production $S \rightarrow S \mid a$. For G we count a, the only word in $L(G)$, infinitely often.

Unfortunately, in general it is undecidable whether a context-free grammar G is ambiguous or not [Aho and Ullman, 1972, p. 203].

The *signature* (of an algebraic specification) is defined as follows [Ehrich *et al.*, 1989, p. 14f]: A *signature* is a pair $\sum =< S, O >$ with S a set of *sorts* and $O = \{O_{\bar{s},s}\}_{\bar{s}\in S^*, s\in S}$ an $S^* \times S$ indexed set family of *operators*. For every word $\bar{s} \in S^*$ and every sort $s \in S$ exists a set $O_{\bar{s},s}$ of operators in O. $\bar{s} = s_1, \ldots s_n$ denotes the list of argument sorts, s the result sort of each operator $o \in O_{\bar{s},s}$. Instead of $o \in O_{\bar{s},s}$ we often write $o : s_1 \times \ldots \times s_n \rightarrow s$. For the case $n = 0$ we obtain the *constants* of sort s which are denoted by $O_{\epsilon,s}$. $X = \{X_s\}_{s\in S}$ is an S indexed set family. The elements of $x \in X_s, s \in S$ are called *variables* of the sort s. We assume that X_s and $O_{\epsilon,s}$ are disjoint for all $s \in S$. This implies that variables and constants are different. With $\sum(X)$ we denote the signature which results from adding all variables as constants to $\sum$ [Ehrich *et al.*, 1989, p. 19]: $\sum =< S, O \cup X >$ with $(O \cup X)_{\epsilon,s} = O_{\epsilon,s} \cup X_s$ and $(O \cup X)_{\bar{s},s} = O_{\bar{s},s}$ for all $s \in S$ and $\bar{s} \in S^+$.

3 COUNTING DERIVATION TREES

In Figure 2 we show, how the set of all complete derivation trees which we can generate in *exactly* 2 derivation steps from the grammar shown in Figure 1 can be counted.

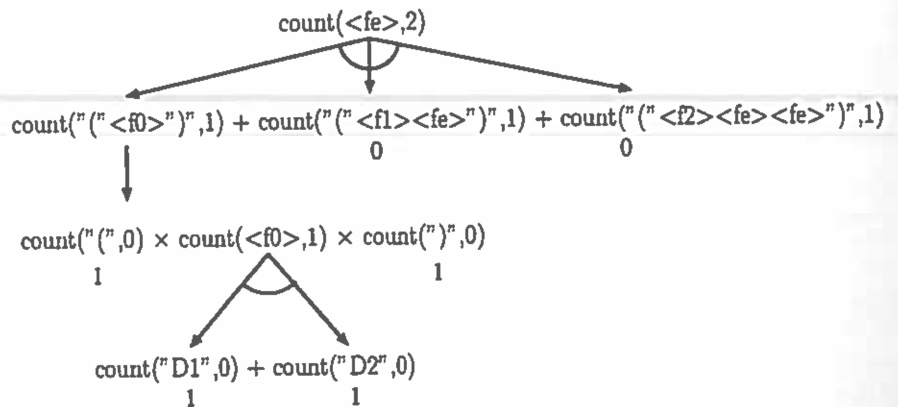

Figure 2: A Small Example

We use a *divide-and-conquer* approach in which the task of counting is successively split into smaller tasks. Starting from the start-symbol <fe> with 2 derivation steps, we write down the set of all possible derivations for this symbol. As a result, we add the results of counting the derivations from the three productions for <fe> with one derivation step. Next, each symbol in the production is assigned the number of derivation steps available. For "("<f0>")" this is easy. The terminal symbols receive 0 derivation steps, the only available derivation step is assigned to <f0>. For simplicity, whenever we allocate derivation steps to symbols, terminal symbols receive 0 derivation steps. The task of counting the number of different trees which can be generated starting from "("<f0>")" can be split into counting the number of trees starting with "(", <f0>, and ")", respectively. In our example, only one assignment of derivation steps is possible. In order to obtain the total number of trees

we take the product of the number of trees starting with "(", <f0>, and ")", because the number of elements of the Cartesian product of the set of trees T_1 starting at position 1 with the set of trees T_2 starting at position 2 and the set of trees T_3 starting at position 3 is $\mid T_1 \mid \times \mid T_2 \mid \times \mid T_3 \mid$. For proof, see [Berge, 1971, p. 16]. Not enough derivation steps are available in the remaining two cases to eliminate all terminals in the derivation, because in both cases the number of nonterminal symbols (2 in the second case and 3 in the third) is higher than the number of available derivation steps (1 derivation step). This implies that some nonterminal symbols remain in the frontier of the derivation tree and that no complete derivation tree could be derived with two derivation steps in both cases. Finally, for <f0> with 1 derivation step we repeat splitting the counting task. The total number of complete derivation trees in the example is 2.

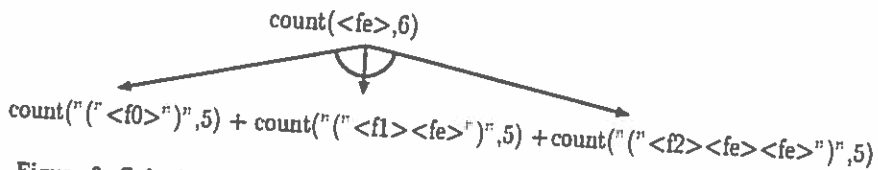

Figure 3: Substitution of <fe>by the Right-Hand Side of the Production Rule

The next example is large enough to illustrate all possible complications in search space counting. Now we count the set of all complete derivation trees which we can generate in exactly 6 derivation steps from the grammar shown in Figure 1. Figure 3 repeats the first step discussed above, this time with 6 derivation steps. Figure 4 shows how terminal symbols are counted, Figure 5 demonstrates the 2-partition case, and Figure 6 illustrates the 3-partition case. In Figure 7 the 3-partition case is derived from applying the 2-partition operation twice.

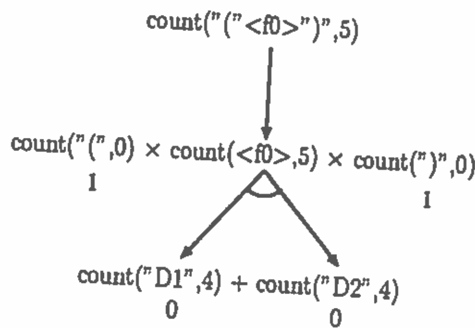

Figure 4: Counting Terminal Symbols (Case 1)

In Figure 4 we have 5 derivation steps which we have to assign to the symbols "(", <f0>, and ")". Terminal symbols do not need any further derivation steps. So, the number of derivation steps assigned to "(" and ")" is 0 and the remaining 5 derivation steps are assigned to <f0>. Because the right hand side of the production rule for <f0> consists of only one terminal symbol, we must assign all available derivation steps to these symbols.

Allocating derivation steps to a terminal symbol indicates that no derivation with exactly 5 derivation steps exists for <f0>. Compare this with the derivation from <f0> with exactly one derivation step shown in Figure 2. Counting terminal symbols with 0 derivation steps assigned results in 1. Counting terminal symbols with 1 or more derivation steps assigned results in 0.

count("("<f1><fe>")",5)

count("(",0) × count(<f1>,1) × count(<fe>,4) × count(")",0)
+
count("(",0) × count(<f1>,2) × count(<fe>,3) × count(")",0)
+
count("(",0) × count(<f1>,3) × count(<fe>,2) × count(")",0)
+
count("(",0) × count(<f1>,4) × count(<fe>,1) × count(")",0)

Figure 5: The 2-Partition Case (Case 2)

In Figure 5 there are 2 nonterminal symbols in the string and 5 derivation steps available. In order to count all complete derivation trees which can be derived in this setting, we have to consider all possible assignments of 5 derivation steps to 2 symbols. All possible ordered 2-partitions (compositions) of 5 are $(1,4)$, $(2,3)$, $(3,2)$, $(4,1)$. These are all pairs (r_1, r_2) for which $r_1 + r_2 = 5$ holds. We have to count the number of complete derivation trees for each of these configurations and to sum over all configurations. Clearly, for a fixed configuration the total number of complete derivation trees is the number of complete derivation trees derivable from the first nonterminal symbol times the number of complete derivation trees derivable from the second.

count("("<f2><fe><fe>")",5)

count("(",0) × count(<f2>,1) × count(<fe>,1) × count(<fe>,3) × count(")",0)
+
count("(",0) × count(<f2>,1) × count(<fe>,3) × count(<fe>,1) × count(")",0)
+
count("(",0) × count(<f2>,3) × count(<fe>,1) × count(<fe>,1) × count(")",0)
+
count("(",0) × count(<f2>,1) × count(<fe>,2) × count(<fe>,2) × count(")",0)
+
count("(",0) × count(<f2>,2) × count(<fe>,1) × count(<fe>,2) × count(")",0)
+
count("(",0) × count(<f2>,2) × count(<fe>,2) × count(<fe>,1) × count(")",0)

Figure 6: The 3-Partition Case (Case 3)

In Figure 6 we have to allocate 5 derivation steps to three nonterminal symbols. Enumeration of the possible configurations results in $(1,1,3)$, $(1,2,2)$, $(1,3,1)$, $(2,1,2)$, $(2,2,1)$, $(3,1,1)$.

In any context-free grammar, at most i nonterminal symbols may occur in the right hand side of a production rule. This implies that at most ordered i-partitions of integers are needed for our task.

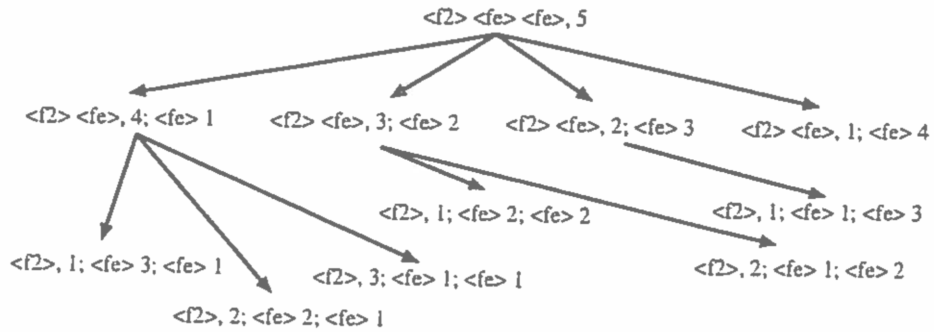

Figure 7: From the 2-Partition Case to the 3-Partition Case

Figure 7 illustrates how we can obtain a 3-partition by repeated applications of 2-partition operations. Because of the associativity of addition, we can combine a fixed collection of 2-partitions in any order we like to obtain the 3-partition. Verification of this for the example in Figure 7 is not included for the sake of brevity. However, we use this fact in generalizing from the 2-partition case in (5) to the k-partition case in (3).

4 FROM THE BNF TO THE WORD COUNTING FUNCTION

The number of words generated by an arbitrary context-free grammar is often infinite. However, we can partition the set of words S into an infinite number of subsets $S_{L,=,i}$. The members of $S_{L,=,i}$ are all words which can be generated in i derivation steps. The set S is then defined by $S = \bigcup_{i=1}^{\infty} S_{L,=,i}$.

Definition 4.1 *A k-bounded search space $S_{L,k}$ contains all words derivable in at most k derivation steps: $S_{L,k} = \bigcup_{i=1}^{k} S_{L,=,i}$*

We denote the size of a search space S by $\mid S \mid$. For a k-bounded search space $S_{L,k}$ the search space is then $\mid S_{L,k} \mid = \sum_{i=1}^{k} \mid S_{L,=,i} \mid$.

Finally, we can characterize the search space size $\mid S_{L,k} \mid$ of L for all $k \in N$ with $\mid S_{L,k} \mid = \sum_{i=1}^{k} \Pi(\langle \text{startsymbol} \rangle, i)$, where $\Pi(\langle \text{startsymbol} \rangle, i)$ denotes a recursive word counting function for $\mid S_{L,=,i} \mid$. (As usual, N denotes the integers $1, 2, 3, \ldots$ and $N_0 = N \cup \{0\}$.)

Definition 4.2 *$\Pi(\langle startsymbol \rangle, i)$ is a recursive word counting function for $\mid S_{L,=,i} \mid$ which is automatically derived from the grammar G of a context-free language L. $\Pi(\langle startsymbol \rangle, i)$ counts the number of words which can be derived from $\langle startsymbol \rangle$ in exactly i derivation steps. (Recursive word counting function.)*

In the following we show in detail how the recursive word counting function is derived from the production rules of the grammar of a context-free language.

For $L(G)$ the recursive word counting function Π can be automatically derived by a signature mapping $F_1 : \sum_{BNF} \rightarrow \sum_{WCF}$ from the productions P of G which are given in Backus-Naur Form. We assume, that all semicolons (rule separators) have already been stripped in a preprocessing phase. F_1 "compiles" a grammar into a word counting function.

$$\sum_{BNF} = \langle V_T, V_{NT}, :=, |, \star \rangle \tag{1}$$

is the signature of the language of the BNF grammar specification, with V_T denoting the terminal alphabet, V_{NT} the nonterminal alphabet, $:=$ denoting *derives to*, $|$ denoting *or* and $\star$ denoting *catenation* (usually catenation is denoted by juxtaposition of symbols). Since $y \in V_{NT}$ may appear on both sides of a production, y_{LHS} indicates its appearance on the left hand side and y_{RHS} its appearance on the right hand side of a production. The signature of the recursive word counting function language is

$$\sum_{WCF} = \langle \Pi(x,d), \Pi(y,d), =, +, \circ \rangle. \tag{2}$$

$\Pi(x,d)$ is the set of all invocations and definitions of the recursive word counting function Π with $x \in V_T$ and $d \in N_0$ as argument. $\Pi(y,d)$ is the set of all invocations and definitions of the recursive word counting function Π with $y \in V_{NT}$ and $d \in N$ as argument. $=$ denotes *is defined by* and $+$ denotes addition. $\circ$ is the ordered 2-partition function explained in Figures 5 to 7 which acts as a kind of a ternary function composition operation. $\circ(\Pi(y_1,d), \Pi(y_2,d), d)$ calculates the number of words derivable from the symbol string $y_1 y_2$ in d derivation steps.

The recursive word counting function scheme $\Pi : (V_T \times N_0) \cup (V_{NT} \times N) \rightarrow N_0$ is now derived by the signature mapping $F_1 : \sum_{BNF} \rightarrow \sum_{WCF}$ which is defined as follows:

1. For all elements x of V_T, x is replaced by a call to $\Pi(x, d-1)$:

   $$\forall x \in V_T : x \rightarrow \Pi(x, d-1) .$$

   We add for all $x \in V_T$ a clause of the following kind to the recursive word counting function scheme Π:

   $$\Pi(x,d) = 1 \cdot (d=0), \qquad \text{where}(S) = \begin{cases} 1 & \text{if } S \text{ is true} \\ 0 & \text{else} \end{cases}$$

   for any statement S which can be either true or false. Therefore, for a terminal symbol x, $\Pi(x,d)$ is 1, if $d = 0$ and 0 otherwise.
2. For all elements y of V_{NT}, the substitution rules for y are given by:

   $$\forall y_{LHS} \in V_{NT} : y_{LHS} \rightarrow \Pi(y_{LHS}, d)$$

   $$\forall y_{RHS} \in V_{NT} : y_{RHS} \rightarrow \Pi(y_{RHS}, r_i)$$

   For a nonterminal symbol y and $d < 1$, Π is undefined. Whenever Π is undefined, it takes the value 0. For r_i see Expression 3.

3. We replace := by =:

$$:= \;\rightarrow\; =$$

4. We replace | by +:

$$| \;\rightarrow\; +$$

5. We replace catenation $*$ by 2-partitions $\circ$:

$$* \;\rightarrow\; \circ$$

In the BNF notation catenation $x_1 * x_2$ is usually formed by writing the symbols immediately together: $x_1 x_2$. Each k-symbol string $y^k = y_1 \ldots y_k$ is therefore replaced by the function $\Gamma(y_1 \ldots y_k, d)$ defined below:

$$\Gamma(y_1 \ldots y_k, d) = \sum_A \prod_{i=1}^{k} \Pi(y_i, r_i), \tag{3}$$

where the summation runs over the set $A = \left\{ r_i \mid \sum_i^k r_i = d - 1, d > 0, r_i \geq 0 \right\}$. Function $\Gamma(y_1 \ldots y_k, d)$ is the ordered k-partition function as shown in Figure 7 for $k = 3$ (3-partitions).

We denote the number of words derivable in d derivation steps from a string $y_1 \ldots y_k$ for one specific partition $r = r_1 \ldots r_k$ of d by

$$\gamma(y_1 \ldots y_k, r) = \prod_{i=1}^{k} \Pi(y_i, r_i) \tag{4}$$

Note that $\Gamma(y_1 \ldots y_k, d) = \sum_A \gamma(y_1 \ldots y_k, r_i)$. Let us return to the string $x_1 x_2$ with the two symbols x_1 and x_2 with e derivation steps available. How many different words can we derive in e derivation steps from the start string $x_1 x_2$? The answer is found after having analysed the following four cases:

(a) Both symbols x_1 and x_2 are nonterminal symbols. In order to derive a string of terminal symbols we have to assign say r_1 derivation steps to x_1. For the partition of e into 2 parts (2-partition) the number of words derivable is the product of the number of words derivable from x_1 in r_1 derivation steps with the number of words derivable from x_2 in r_2 derivation steps. To find all words, we have to sum over all 2-partitions. By all ordered k-partitions of d derivation steps, we mean all k-tuples of integers $r_1, \ldots, r_k$ such that $r_1 + r_2 + \ldots + r_k = d$, with $r_i \geq 0$. This means, all partitions generated by pairs r_1, r_2 such that $r_1 + r_2 = e$ and $r_1, r_2 \neq r_2, r_1$ hold:

$$\circ(\Pi(x_1, e), \Pi(x_2, e), e) = \sum_{\substack{r_1 + r_2 = e, r_1 \geq 0, \\ r_2 \geq 0, e \geq 0}} \Pi(x_1, r_1) \cdot \Pi(x_2, r_2) \tag{5}$$

If $r_i = 0$, $\Pi(x_i, r_i)$ is undefined, because we cannot derive a string of terminal symbols and this implies that $\Pi(x_i, r_i) = 0$. For further information regarding the theory of partitions of integers see [Andrews, 1976].

(b) x_1 is a terminal symbol and x_2 a nonterminal symbol. With Expression 5 we obtain again our result. You can convince yourself that this is the case, because $\Pi(x_1, r_1)$ is 0 for all values of r_1 except for $r_1 = 0$.

(c) x_1 is a nonterminal symbol and x_2 a terminal symbol. The roles of x_1 and x_2 have changed. With Expression 5 we obtain again our result. You can see for yourself that this is the case, because $\Pi(x_2, r_2)$ is 0 for all values of r_2 except for $r_2 = 0$.

(d) Both symbols x_1 and x_2 are terminal symbols. Expression 5 is only defined, if e, r_1 and r_2 are 0. In this case, its result is 1.

In order to obtain Function 3, we only have to generalize Expression 5 to the n-partition case, taking terminal symbols into account. Obviously, by doing this we have generalized the $2+1$-ary operation $\circ$ to a $n+1$-ary operation.

For example, by applying the signature mapping F_1 to the production rules for L_{XOR} shown in Figure 1 we obtain the recursive function shown in (6). Note, that we have folded the clauses for the terminal symbols into an "otherwise" clause.

$$\Pi(y,d) = \begin{cases} \Gamma(\texttt{"("<f0>")"}, d-1) + \Gamma(\texttt{"("<f1><fe>")"}, d-1) + & \\ \quad \Gamma(\texttt{"("<f2><fe><fe>")"}, d-1) & \text{if } y = \texttt{<fe>} \\ \Gamma(\texttt{"D1"}, d-1) + \Gamma(\texttt{"D2"}, d-1) & \text{if } y = \texttt{<f0>} \\ \Gamma(\texttt{"NOT"}, d-1) & \text{if } y = \texttt{<f1>} \\ \Gamma(\texttt{"OR"}, d-1) + \Gamma(\texttt{"AND"}, d-1) & \text{if } y = \texttt{<f2>} \\ 1 \cdot (d = 0) & \text{otherwise} \end{cases} \tag{6}$$

In Table 2 we show the search space sizes of $S_{L_{XOR},=,i}$ up to $i = 24$ and the probability that a word is drawn from $S_{L_{XOR},=,i}$ if we draw with equal probability from all words in $S_{L_{XOR},24}$. To increase the performance of the search space counting functions we recommend to use the corresponding memoizing functions [Geyer-Schulz, 1989]. A memoizing function computes the value of a function for an argument the first time it is called with this argument and stores the value in a lookup table. For all other function calls with this argument the memoizing functions return the value in the lookup table.

Table 2: The Search Space Sizes $S_{L_{XOR},=,i}$

Derivation Steps i	$\lvert S_{L_{XOR},=,i} \rvert$	$P(X \in S_{L_{XOR},=,i})$
2	2	0.0000006
4	2	0.0000006
6	10	0.0000032
8	26	0.0000082
10	114	0.0000360
12	402	0.0001269
14	1722	0.0005435
16	6890	0.0021745
18	29794	0.0094032
20	126626	0.0399640
22	556778	0.1757227
24	2446138	0.7720167

5 AN EXACT UNIFORM INITIALIZING ALGORITHM

In Table 1 we observed an exponential decay in the probability of generating words, as the number of derivation steps n increased. However, in the following we develop an exact, uniform sampling algorithm for initializing populations of words of k-bounded context-free languages. The algorithm has two phases:

1. From the sets $S_{L,=,i}$ of the search space of a k-bounded context-free language draw a partition $S_{L,=,i}$ to which the word we intend to generate belongs. (Draw a partition of the search space.)
2. Generate a derivation tree with exactly i derivation steps. (Generate a derivation tree.)

Draw a partition of the search space. In the first step of the exact uniform initialization algorithm we determine the partition $S_{L,=,i}$ to which the word we intend to generate should belong. The probability of drawing a complete derivation tree with (exactly) i derivation steps from start symbol S in a k-bounded context-free language L with grammar G with equal probability is trivially given as:

$$P(\text{tree in } i \text{ derivation steps}) = \frac{\Pi(S,i)}{\sum_{i=0}^{k} \Pi(S,i)} \qquad (7)$$

For the grammar L_{XOR} bounded to 24 derivation steps the probability of drawing a word derivable in 20 derivation steps is 0.0399640. Column 3 of Table 2 shows the probability of drawing a word from partition $S_{L,=,i}$. Table 2 has been computed with the help of (6).

Generate a derivation tree. The function INIT_WORD_U (see Figure 8) implements an algorithm for generating a complete derivation tree from start symbol S in exactly i derivation steps with equal probability. However, for ease of use e.g. as part of other genetic operators, if no word can be derived in d derivation steps, a word from the non-empty partition $S_{L,=,i}$ with the largest $i < d$ is generated. In the initialization algorithm an invocation of the algorithm INIT_WORD_U on empty sets $S_{L,=,i}$ is not possible, because empty sets $S_{L,=,i}$ have probability 0. The algorithm consists of

1. a recursive tree generation algorithm without backtrack,
2. a randomized choice function for selecting the appropriate production rule for expansion,
3. and a randomized choice function for selecting the appropriate k-partition of derivation steps. This function assigns to each symbol in a selected production rule the number of derivation steps which are available for expanding this symbol on the next level of recursion.

The word generating function. In Figure 8 we present the pseudo-code for the word generating function.

In the pseudo-code we assume the existence of a generic list data type with the operations *new_list*, *add_list*, *head*, *tail* and the predicate *empty*. *new_list* generates an empty list, *add_list* appends its second argument to the list given as its first argument and returns the result, *head* returns the first element of the list given as argument, *tail* returns the list given

```
tree function INIT_WORD_U(symbol root, int d);
   int production;
   list_of_symbol production_symbols;
   list_of_int partition;
   list_of_trees subtree_list;
begin
   if TERMINAL(root) then
      begin
         return(new_tree(root, new_list))
      end
   else
      begin
         production := choose_production(access_ST(root), d-1);
         production_symbols := access_PT(production);
         partition := choose_partition(production_symbols, d-1);
         subtree_list := new_list;
         while not empty(production_symbols) do
            begin
               subtree_list := add_list(subtree_list,
                  INIT_WORD_U(head(production_symbols),head(partition)));
               production_symbols := tail(production_symbols);
               partition := tail(partition);
            end
         return(new_tree(root, subtree_list))
      end
end
```

Figure 8: Pseudocode for the Word Generating Function

as argument without first element. The predicate *empty* returns *true* if its list argument is empty.

The derivation tree is of the form (*root*, *list of subtrees*). The only operation we need is *new_tree* which generates a new derivation tree with its first argument as *root* and its second argument as *list of subtrees*.

The predicate *TERMINAL* returns *true* if its argument is a terminal symbol of the grammar.

The access functions *access_ST* and *access_PT* access the symbol table *ST* shown in Figure 9 and the production table shown in Figure 10, respectively. Both figures are explained at the end of this section.

The choice function for production rules (choose_production). Suppose a nonterminal symbol y can be substituted by p_j, the right hand side of a production rule, $j = 1, \ldots, m$. Of course, each p_j is a k-symbol string $y_1 \ldots y_k$. If we know how many words can be generated starting from each string p_j, we can easily compute the probability to choose p_j so that we draw a complete derivation tree with equal probability. See (3).

$$P(p_j \text{ in } d \text{ derivation steps}) = \frac{\Gamma(p_j, d)}{\sum_{j=1}^{m} \Gamma(p_j, d)} \quad (8)$$

The choice function for k-partitions (choose_partition). For each symbol y_i in the right hand side of a production rule $y_1 \dots y_k$ we have to assign the number r_i of derivation steps available for its expansion, so that $d = \sum_{i=1}^{k} r_i$. We call the vector r a k-partition of d. The number of complete derivation trees which can be generated in d derivation steps starting from $y_1 \dots y_k$ with k-partition r is $\gamma(y_1 \dots y_k, r) = \prod_{i=1}^{k} \Pi(y_i, r_i)$. See (4). The probability of choosing the k-partition r, so that complete derivation trees are generated with equal probability, is:

$$P(r) = \frac{\gamma(y_1 \dots y_k, r)}{\Gamma(y_1 \dots y_k, d)} \quad (9)$$

The symbol table used is shown in Figure 9. The first three columns contain the symbol name, a 1 for nonterminals, and the numerical identifier used in the implementation.

```
<fe>   1   1   1 2 3
<f0>   1   2   4 5
<f1>   1   3   6
<f2>   1   4   7 8
(      0   5
)      0   6
D1     0   7
D2     0   8
NOT    0   9
OR     0  10
AND    0  11
```

Figure 9: The Symbol Table ST

The production table is shown in Figure 10. The first column contains the left hand side of a production, the second the right hand side.

```
<fe>   ( <f0> )
<fe>   ( <f1> <fe> )
<fe>   ( <f2> <fe> <fe> )
<f0>   D1
<f0>   D2
<f1>   NOT
<f2>   OR
<f2>   AND
```

Figure 10: The Production Table in Symbolic Form

6 EXPERIMENTAL RESULTS

In order to test the implementation of the exact uniform initializing algorithm developed above we constructed the following experiments.

The simplest experiment to test the correctness of the implementation is to generate a large number of trees with a sufficiently small number of derivation steps and to test whether the trees are generated with equal probability by comparing the expected number of occurrences of each tree with the observed number in the experiment. We generated a population of $n = 1540$ complete derivation trees with at most 10 derivation steps for the grammar shown in Figure 1. Because at most 154 different complete derivation trees can be generated, under the hypothesis that complete derivation trees are generated with equal probability, we expect each complete derivation tree to occur 10 times in the population. In this experiment and in the three other experiments of this section we performed a standard χ^2-test of goodness of fit. See e.g. [Bhattacharyya and Johnson, 1977, pp 424]. For a confidence level of $\alpha = 0.05$ this hypothesis is accepted ($\chi^2 = 150.8$, $\chi^2 < c = 182.49$, degrees of freedom $DF = 153$).

However, for search spaces bounded with a larger number of derivation steps this approach is infeasible because of the rapid growth of the number of trees in the search space for most grammars. See Table 2. For the grammar shown in Figure 1 we generated a population of $n = 2000$ complete derivation trees with at most 30 derivation steps and we tested the following hypotheses:

1. The complete derivation trees are distributed uniformly over the $S_{L,=,i}$ with $i = 1,\ldots,30$. The result of this experiment is shown in Table 3. Note that the number of derivation trees with an odd number of derivation steps is zero. At a confidence level of $\alpha = 0.05$ we accept this hypothesis ($\chi^2 = 4.67$, $\chi^2 < c = 7.81$, degrees of freedom $DF = 3$).

Table 3: Distribution of Derivation Trees over Number of Derivation Steps

Derivation Steps	Expected	Observed
< 24	22.22	23
26	76.65	92
28	343.81	361
30	1557.32	1524

2. The complete derivation trees are distributed uniformly over the following partition: Class 1 contains all complete derivation trees starting with <fe> ⇒ "(" <f0> ")" or <fe> ⇒ "(" <f1> <fe> ")" and class 2 contains all derivation trees starting with <fe> ⇒ "(" <f2> <fe> <fe> ")". The results of this experiment are shown in Table 4. Again, at a confidence level of $\alpha = 0.05$ this hypothesis is accepted ($\chi^2 = 1.36$, $\chi^2 < c = 3.84$, degrees of freedom $DF = 1$).

3. We further partitioned the classes according to length. Table 5 shows the results. Again, we accept the hypothesis that the complete derivation trees are uniformly distributed at a confidence level of $\alpha = 0.05$ ($\chi^2 = 6.33$, $\chi^2 < c = 14.07$, degrees of freedom $DF = 7$).

Table 4: Distribution of Derivation Trees over Class 1 and Class 2

Class	Expected	Observed
1	442.68	421
2	1557.32	1579

Table 5: Distribution of Derivation Trees over Classes and Derivation Steps

Class	Derivation Steps	Expected	Observed
1	< 24	5.07	5
1	26	17.15	20
1	28	76.65	73
1	30	343.81	323
2	< 24	17.15	18
2	26	59.49	72
2	28	267.16	288
2	30	1213.50	1201

7 THE IMPACT ON GENETIC PROGRAMMING PERFORMANCE

To show the impact of our exact uniform initialization procedure on genetic programming performance we tried an experiment. We repeated the genetic algorithm with each initialization algorithm 100 times for a population size of 50, a k-bound of 40 and with at most 50 generations with a mutation rate of 0.05, a crossover rate of 0.7 and elite selection on the XOR-problem.

The biased genetic algorithm succeeded 45 times in finding a correct solution for the XOR-function, the exact uniform genetic algorithm succeeded 61 times. A statistical comparison of these success rates is most conveniently performed by means of a 2×2 contingency table:

	Success	No Success
Biased GA	45	55
Uniform GA	61	39

The corresponding χ^2 test statistic is $T = 5.13$ which exceeds the critical value $c = 3.84$, $(DF = 1)$ at a significance level of $\alpha = 0.05$. For details see [Bhattacharyya and Johnson, 1977, pp 440].

Similar improvements are reported by Iba for his heuristic random tree generation algorithm for Koza's genetic programming variant for learning boolean functions with 3 arguments [Iba, 1996a], simple symbolic regression, trigonometric identities, and predicting a Mackey-Glass time series [Iba, 1995].

However, speculation based on a few simulation results always remains unsatisfactory. There is nothing more practical (and final) than a proof of what constitutes the best initialization algorithm for the case that no a priori information on the solution is available.

Because of the stochastic nature of genetic programming algorithms, their initialization is essentially a problem of statistical sampling theory. There are several ways to devise a rule or *sampling design* which determines the probability that a particular point of the search space is included into the initial population or not. One such rule is uniform initialization or *simple random sampling*, which assigns to each point in the search space the same probability of being sampled. An alternative would be stratified sampling which is based on a partition of the search space according to some reliable a priori information about the possible locations of the optimum. Here points belonging to different parts or strata are assigned different probability of being included into the initial population. Several other sampling designs have been proposed in statistics, the interested reader is referred to [Krishnaiah and Rao, 1988] for further details.

The decision which design to use should be certainly oriented on some measure of efficiency or optimality. Interestingly, it turns out, that simple random sampling is optimal in the sense of being a *minimax procedure*, provided that we do not have any useful a priori information about the possible locations of the optimum in search space. The result we are going to present is not new, it has in fact been known since 1954 and comes from [Blackwell and Girshick, 1954, Chapter 8].

In the sequel we will outline the basic ideas of Blackwell and Girshick, suitably adapted, however, to the situation we encounter when initializing genetic programming algorithms.

Let S denote the search space which we assume to be finite with cardinality M and define X to be the set containing S and all its permutations. Let the triple $(X; \Omega, p)$ denote a sample space in the sense of Blackwell and Girshick, where Ω is a parameter space being equal to X and p is the trivial probability measure

$$p(x|\omega) = \begin{cases} 1 & \text{if} \quad x = \omega \\ 0 & \text{if} \quad x \neq \omega. \end{cases}$$

Next let us define a space of actions A in the following way: A contains all possible initial populations of size $N < M$ which may be sampled from X, with the proviso that there are no duplicate individuals in $a \in A$. Thus we consider sampling designs without replacement. Observe that a violation of this restriction is very unlikely to have serious effects, as long as M is large compared to N, a situation which we typically encounter in practice. This point can be made more precise, see for instance [Pathak, 1988]. The action associated with $a \in A$ is simply: run the genetic programming algorithm GP with initial population a.

Let V denote a set of sampling designs which consists of all subsets of the form

$$v = (j_1, j_2, \ldots, j_N)$$

of the integers $1, 2, \ldots, M$, such that $N < M$ and all components in v are different. Choosing a particular $v \in V$ means, create the initial population consisting of the individuals $j_1, j_2, \ldots, j_N$.

On V we define a decision function which is the identity, i.e. $a = v$, where equality means, that a and v contain the same individuals regardless of order. The decision is: if v has been drawn from X, then run the genetic programming algorithm GP with initial population a.

Let $\phi(v, a|x)$ be a randomized procedure, i.e. a probability measure defined on $V \times A \times X$ as

$$\begin{aligned} \phi(v, a|x) &= P(\text{select } v \text{ and start GP with population } a) \\ &= \begin{cases} \pi(v) & \text{if } \quad v = a \\ 0 & \quad\quad \text{else} \end{cases}, \end{aligned} \tag{10}$$

where $\pi(v)$ is the probability that sample v is drawn. Observe that $\pi(v)$ is just the sampling design, which we are looking for. If $\pi(v)$ is equal for all possible samples v, then we have simple random sampling.

Note that $\sum_v \pi(v) = 1$ and observe also that by (10)

$$\sum_{v \in V} \sum_{a \in A} \phi(v, a|x) = 1,$$

furthermore, $\phi(v, a|x)$ depends only on the individuals of x contained in v and we assume that

$$\sum_{a \in A} \phi(v, a|x)$$

is independent of x for all $v \in V$.

To any particular choice of v and corresponding action a we associate a certain loss L, which is conveniently measured by the time complexity of the genetic programming algorithm GP initialized by population a. The loss function L is defined on $\Omega \times A$ and is assumed to be constant with respect to permutations of $\omega = x$. More formally we write

$$L(\omega, a) = L(\omega, (v, a)).$$

Let $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_M)$ denote a permutation of $1, 2, \ldots, M$, and let

$$\sigma(x) = (x_{\sigma_1}, \ldots, x_{\sigma_M})$$

and if $v = (j_1, j_2, \ldots, j_N)$, then define

$$\sigma(v) = (\sigma_{j_1}, \sigma_{j_2}, \ldots, \sigma_{j_N}).$$

Also let $\sigma(v, a) = (\sigma(v), a)$.

Now observe that the loss function is trivially invariant with respect to permutations, i.e.

$$L(\omega, (v, a)) = L(\sigma(\omega), \sigma(v, a)),$$

and note also that

$$p(\sigma(x)|\sigma(\omega)) = 1 \quad \text{if} \quad \sigma(x) = \sigma(\omega)m$$

and thus $x = \omega$ and $p(x|\omega) = 1$. Moreover

$$p(\sigma(x)|\sigma(\omega)) = 0 \quad \text{if} \quad \sigma(x) \neq \sigma(\omega),$$

and therefore $x \neq \omega$ and $p(x|\omega) = 0$.

The risk associated with the random procedure $\phi(v, a|x)$, i.e. the risk induced by starting the genetic programming algorithm GP with a particular initial population, will be denoted by $\rho(\omega, \phi)$ and is defined by

$$\rho(\omega, \phi) = \sum_{x \in X} \sum_{a \in A} L(\omega, (v, a))\phi(v, a|x)p(x|\omega).$$

Obviously, the risk function is invariant with respect to permutations σ applied to both, ω and (v, a). It follows from theorem 8.6.4 of Blackwell and Girshick (1954), that there exists a minimax procedure ϕ^*, such that

$$\sup_{\omega \in \Omega} \rho(\omega, \phi^*) \leq \sup_{\omega \in \Omega} \rho(\omega, \phi).$$

Furthermore, we have by invariance that

$$\sum_{a \in A} \phi^*(v, a|x) = \sum_{a \in A} \phi^*(\sigma(v), a|\sigma(\omega)),$$

and both sides are independent of x by definition and equal to $\pi(v)$, the probability of selecting sample v.

Thus it follows that

$$\pi(v) = \pi(\sigma(v)).$$

If we sum both sides of this equation over all $M!$ permutations, then we obtain

$$M!\pi(v) = (M - N)!,$$

since $\sum_v \pi(v) = 1$ and since there are to each sample v $N!$ permutations which represent the same initial population. Hence we finally get

$$\pi(v) = 1/\binom{M}{N}. \quad (11)$$

However, equation (11) states, what the minimax sampling design is: draw samples uniformly or apply simple random sampling.

These results are in accordance with theoretical results of [Strasser, 1978] and [Strasser, 1976] which prove in a (Bayesian) setting that choosing the non-informative prior distribution improves the speed of convergence of a Bayesian learning algorithm.

In addition, sampling theory (e.g. [Hansen *et al.*, 1953]) requires simple random sampling for the usual sample statistics to be unbiased estimators of the population statistics. In a Bayesian setting simple random sampling constitutes the obvious noninformative prior or the "most uncertain" or maximum entropy prior [Berger, 1988].

8 EXACT UNIFORM INITIALIZATION FOR KOZA'S GENETIC PROGRAMMING VARIANT

Because of recent interest and a large number of applications of Koza's genetic programming variant ([Koza *et al.*, 1996] and [Koza, 1996]), it is desirable to apply the algorithms

developed above to Koza's genetic programming variant too. By using a well-known mapping between signatures of algebraic specifications to context-free grammars (which is the foundation of the algebraic semantic of programming languages) we show that we can express Koza's specification of programming languages in terms of a "terminal set" (parse tree leaves) and a "nonterminal set" (interior nodes of a parse tree) in terms of a signature $\sum(X)$ and that this signature can be transformed into a context-free grammar.

Next, we show how Koza style specifications of programming languages can be converted into a signature $\sum(X)$. Because of the closure property of Koza's genetic programming variant (all functions must accept as arguments the results of any other functional expression of the language) the signature has only one sort, namely *functional expressions*. Koza's "terminal set" corresponds to the set of variables and constants $O_{\epsilon,s} \cup X_s$ of the sort and the "nonterminal set" corresponds to the set of operators O.

Now we are again on familiar ground. For each signature $\sum(X) =< S, O \cup X >$ we obtain an S-indexed set family $G(\sum) = \{G(\sum)_{s_0} = (N, T, s_0, P)\}_{s_0 \in S}$ of context-free grammars by the following mapping [Ehrich *et al.*, 1989, p. 16]:

1. The nonterminals are $N = S$.
2. The terminal are $T = \overline{O} \cup \{(,)\} \cup \{,\}$ with $\overline{O}$ the disjoint union of all sets $O_{\overline{s},s}$ for $\overline{s} \in S^*$ and $s \in S$.
3. The start symbol of $G(\sum)_{s_0}$ is s_0.
4. The productions are defined by $P = \{s \rightarrow o(s_1, \ldots, s_n) \mid o \in O_{s_1 \ldots s_n, s}\}$.

For every context-free grammar $G = (N, T, P, X_0)$ a signature $\sum(G) =< S, O >$ can be allocated as follows:

1. The sorts are $S = N$.
2. The operators in $O_{\overline{s},s}$ are the productions with left hand side s and the string $\overline{s}$ of nonterminal symbols on the right hand side.

As a corollary we immediately see that the programming languages used in Koza's genetic programming variant form a proper subset of context-free languages, namely those context-free languages which can be generated with exactly one nonterminal symbol.

In our practical example we use a (slightly) modified version of the mapping from signatures to context-free grammars:

1. The nonterminals are $N = S$.
2. The terminal are $T = \overline{O} \cup \{(,)\}$ with $\overline{O}$ the disjoint union of all sets $O_{\overline{s},s}$ for $\overline{s} \in S^*$ and $s \in S$.
3. The start symbol of $G(\sum)_{s_0}$ is s_0.
4. The productions are defined by $P = \{s \rightarrow (os_1 \ldots s_n) \mid o \in O_{s_1 \ldots s_n, s}\}$. (For those reader who prefer PostScript (an interpreter with postfix notation) we change P to $P = \{s \rightarrow (s_1 \ldots s_n o) \mid o \in O_{s_1 \ldots s_n, s}\}$.)

To apply the exact uniform initialization algorithm presented in Section 5 all what remains to be done is to explicitly derive the context-free grammar $G(\sum)_s$ induced by the signature $\sum(X)$ which is implicitly defined by Koza's genetic programming variant.

For example, [Iba, 1996b] considers the following specification of a language in Koza's style by defining a "terminal set" T and a "nonterminal set" F. The subscripts of a "nonterminal" represents the arity of the "nonterminal":

$$T = \{D_0, D_1, D_2, D_3\} \tag{12}$$
$$F = \{\mathrm{AND}_2, \mathrm{OR}_2, \mathrm{NAND}_2, \mathrm{NOR}_2\} \tag{13}$$

From (13) we can easily see that all functions are of arity 2. Let A be the arity set which in this case is $A = \{2\}$. By applying the mapping presented above to Iba's example, we obtain the grammar shown in Backus Naur form in Figure 11.

```
S := <fe> ;
<fe> := "(" "D0" ")" | "(" "D1" ")" | "(" "D2" ")" | "(" "D3" ")" |
        "(" "AND" <fe> <fe> ")" | "(" "OR" <fe> <fe> ")" |
        "(" "NAND" <fe> <fe> ")" | "(" "NOR" <fe> <fe> ")" ;
```

Figure 11: The Backus Naur Form for Iba's Example

This mapping enables us to compare the exact uniform initialization algorithm with the random tree generation heuristic developed by Iba in [Iba, 1995] and [Iba, 1996b] for Koza's genetic programming variant. Iba treats Koza's parse trees as unlabeled structures.

Table 6: Counting Labeled and Unlabeled Trees

n	Catalan(n)	Unlabeled, $A = \{2\}$	Labeled
1	1	1	4
2	1	0	0
3	2	1	64
4	5	0	0
5	14	2	2048
6	42	0	0
7	132	5	81920
8	429	0	0
9	1430	14	3670016
10	4862	0	0

As a first estimation for the number of parse trees with n nodes for a language he proposes the n-th Catalan number

$$Cn(n) = \frac{1}{n}\binom{2n-2}{n-1}$$

which counts the number of unlabeled trees with n nodes. This is shown in column 2 of Table 6. However, not every unlabeled tree is a valid parse tree for Iba's language. In [Iba, 1995] a method of counting all unlabeled trees which respect the arity constraints is presented. The number of these trees can be shown to be

$$\frac{1}{n}\binom{n}{\frac{n-1}{2}}$$

for Iba's example with the proviso that this binomial coefficient is 0, whenever n is an even number. See Table 6, column 3. The general case is covered in [Goulden and Jackson, 1983, pp. 111]. The last column in Table 6 gives the number of derivation trees for the language specified by (12) and (13).

The interested reader will certainly recognize a striking difference between columns 3 and 4 of Table 6. This discrepancy can be explained by the simple fact that derivation trees are labeled ordered trees for context-free grammars. As we have shown above, the language specified in (12) and (13) gives rise to the context-free grammar shown in Figure 11.

9 CONCLUSION

The algorithms in this paper solve the problem of exact uniform generation of complete derivation trees from k-bounded context-free languages. They are the basis for an exact uniform initialization routine for simple genetic algorithms over k-bounded context-free grammars, a variant of genetic programming. It is important to note that, in principle, uniform initialization may be used as a starting point for the design of initialization algorithms for genetic programming which utilize a priori information about the search space and thus require biased initialization procedures. However, it seems that the practical implementation of these algorithms is, in general, highly non trivial, because such issues like balance of trees, depth, number of inner nodes, ...have to be taken into account. Furthermore, the result of Section 4 gives prerequisites for the combinatorial analysis of crossover and mutation operators on complete derivation trees. This is of crucial importance for deriving a schema-theorem, the corresponding loss function, and for the comparison with other stochastic optmization methods. This task is left for further research.

The complexity of our algorithm depends on the following:

1. The signature mapping $F1$ which compiles a grammar into a word counting function has complexity O(number of the symbols in the grammar).
2. For the tabulation of $\Pi(y,d)$, $\Gamma(y_1 \dots y_k, d)$ and $\gamma(y_1 \dots y_k, d)$ the complexity is dependent on the recursive structure of the grammar and on the number of ordered partitions (compositions) including zeroes $c(k,d)$. See [Andrews, 1976, p. 54].

$$c(k,d) = \binom{d+k-1}{k-1} \sim O(d^k), \text{ for fixed } k \text{ and } d \to \infty.$$

For most of the grammars used in practical applications of genetic programming, the largest k is very small, because function arities are seldom larger than 4. However, there is still hope that further complexity reductions can be achieved with the help of normal-form theory and formal power series techniques. For example, each context-free grammar can be transformed to Chomsky normal form [Aho and Ullman, 1972, p. 151f]. In this case k will be equal to 2 and the number of ordered partitions is linear in the number of derivation steps. However, currently we do not know how the

transformation to Chomsky normal form affects the recursive structure of the grammar. This requires further research.

3. The complexity of the word generating function shown in Figure 8 is of $O(d)$, the number of derivation steps, provided $\Pi(y, d)$, $\Gamma(y_1 \dots y_k, d)$ and $\gamma(y_1 \dots y_k, d)$ have been tabulated.

Acknowledgements

We gratefully acknowledge the help of Hitoshi Iba, William B. Langdon, Peter Whigham, and four anonymous referees. Their comments and suggestions as well as their help in getting papers otherwise inaccesible to us over the internet have considerably improved this work.

References

[Aho and Ullman, 1972] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling, Volume I: Parsing*, volume 1. Prentice–Hall, Inc., Englewood Cliffs, N.J., 1972.

[Andrews, 1976] George E. Andrews. *The Theory of Partitions.* Addison Wesley, Reading, 1976.

[Berge, 1971] Claude Berge. *Principles of Combinatorics*, volume 72 of *Mathematics in Science and Engineering.* Academic Press, New York, 2nd edition, 1971.

[Berger, 1988] James O. Berger. *Statistical Decision Theory and Bayesian Analysis.* Springer Series in Statistics. Springer Verlag, New York, 2nd edition, 1988.

[Bhattacharyya and Johnson, 1977] G. K. Bhattacharyya and R. A. Johnson. *Statistical Concepts and Methods.* Wiley, New York, 1977.

[Blackwell and Girshick, 1954] David Blackwell and M. A. Girshick. *Theory of Games and Statistical Decisions.* John Wiley & Sons, New York, 1954.

[Brown *et al.*, 1988] James A. Brown, Sandra Pakin, and Raymond P. Polivka. *APL2 at a Glance.* Prentice–Hall, Inc., Englewood Cliffs, N.J., 1988.

[Ehrich *et al.*, 1989] Hans-Dieter Ehrich, Martin Gogolla, and Udo Walter Lipeck. *Algebraische Spezifikation abstrakter Datentypen.* Leitfäden und Monographien der Informatik. B. G. Teubner, Stuttgart, 1989.

[Geyer-Schulz, 1989] Andreas Geyer-Schulz. Memo. *APL Quote Quad*, 20(2):12–27, December 1989. ACM, New York.

[Geyer-Schulz, 1995] Andreas Geyer-Schulz. *Fuzzy Rule-Based Expert Systems and Genetic Machine Learning*, volume 3 of *Studies in Fuzziness.* Physica-Verlag, Heidelberg, 1995.

[Geyer-Schulz, 1996a] Andreas Geyer-Schulz. Compound derivations in fuzzy genetic programming. *Proc. NAFIPS'96*, pages 510–514, July 1996.

[Geyer-Schulz, 1996b] Andreas Geyer-Schulz. *Fuzzy Rule-Based Expert Systems and Genetic Machine Learning*, volume 3 of *Studies in Fuzziness and Soft Computing.* Physica-Verlag, Heidelberg, 2nd revised edition, 1996.

[Goulden and Jackson, 1983] Ian P. Goulden and David M. Jackson. *Combinatorial Enumeration*. John Wiley & Sons, New York, 1983.

[Gupta *et al.*, 1994] Rajiv Gupta, Scott A. Smolka, and Shaji Bhaskar. On randomization in sequential and distributed algorithms. *ACM Computing Surveys*, 26(1):7–86, March 1994.

[Hansen *et al.*, 1953] Morris H. Hansen, William N. Hurwitz, and William G. Madow. *Sample Survey Methods and Theory*. John Wiley & Sons, New York, 1953.

[Iba, 1995] Hitoshi Iba. Random tree generation for genetic programming. Technical Report ETL-TR-95-35, Electrotechnical Laboratory (ETL), 1-1-4 Umezono, Tsukuba Science City, Ibaraki, 305, Japan, November 14th 1995.

[Iba, 1996a] Hitoshi Iba. Random tree generation for genetic programming. In Koza [1996], pages 75–82.

[Iba, 1996b] Hitoshi Iba. Random tree generation for genetic programming. *Parallel Problem Solving From Nature (PPSN'96)*, 1996. to appear.

[IBM, 1985] IBM. *APL2 Programming: Language Reference*. IBM Corporation, San Jose, 1985.

[Koza *et al.*, 1996] John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors. *Genetic Programming 1996: Proceedings of the First Annual Conference*, Cambridge, MA, July 1996. MIT Press.

[Koza, 1992] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, Massachusetts, 1992.

[Koza, 1996] John R. Koza, editor. *Late Breaking Papers at the Genetic Programming 1996 Conference*, Stanford, July 1996. Stanford University Bookstore.

[Krishnaiah and Rao, 1988] P. R. Krishnaiah and C. R. Rao, editors. *Handbook of Statistics: Sampling*, volume 6, Amsterdam, 1988. North-Holland.

[Kuich and Salomaa, 1986] Werner Kuich and Arto Salomaa. *Semirings, Automata, Languages*, volume 5 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, Berlin, 1986.

[Naur, 1963] P. Naur. Revised report on the algorithmic language ALGOL 60. *Communication of the ACM*, 6(1):1–17, 1963.

[Pathak, 1988] P. K. Pathak. Simple random sampling. In Krishnaiah and Rao [1988], pages 97–110.

[Rosca, 1995] J. Rosca, editor. *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, San Mateo, July 1995. Morgan Kaufmann.

[Strasser, 1976] Helmut Strasser. Asymptotic properties of posterior distributions. *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, 35:269–282, 1976.

[Strasser, 1978] Helmut Strasser. Admissible representations of asymptotically optimal estimates. *The Annals of Statistics*, 6(4):867–881, 1978.

[Whigham, 1995a] Peter A. Whigham. Grammatically-based genetic programming. In Rosca [1995], pages 33–41.

[Whigham, 1995b] Peter A. Whigham. Inductive bias and genetic programming. In Zalzala [1995], pages 461–466.

[Whigham, 1996] Peter A. Whigham. Search bias, language bias, and genetic programming. In Koza et al. [1996], pages 230–237.

[Zalzala, 1995] A. M. S. Zalzala, editor. *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, volume 414, London, September 1995. IEE.

A THE APL COMPILER

As executable notation which allows the reader immediate experimentation we add the APL2 implementation of the algorithms presented in the paper. For an introduction to APL2 we refer the interested reader to [Brown *et al.*, 1988] and [IBM, 1985]. The APL2 source code is available from `http://mortadelo.wu-wien.ac.at/usr/genetic/`.

The signature mapping $F1$ is implemented by the APL function PI_COMP which takes a BNF as argument and returns the canonical representation of the APL function PI which implements the recursive word counting function $\Pi(y, d)$ shown in (6). The function PI_COMP calls the function BNF_COMP given in [Geyer-Schulz, 1996b, p. 242f] which returns a four column symbol table ST (symbols of the language, nonterminal/terminal symbol, symbol identifier, list of indices of symbol in production table PT), a two column production table (symbol identifier, production) and the identifier of the start symbol START.

```
    ∇Z←PI_COMP BNF;ST;PT;START;HEAD;NT_CODE;T_CODE;NT
[1]   (ST PT START)←BNF_COMP BNF
[2]   HEAD←'Z←PI A;X;D' '(X D)←A' '⍙''→LABEL'',⍕X' 'Z←''
      ERROR''' '→0'
[3]   T_CODE←'Z←1×(D=0)' '→0'
[4]   T_CODE←(((⊂'LABEL'),¨(⍕¨(0=ST[;2])/ST[;3])),¨':'),T_CODE
[5]   NT_CODE←(((⊂'LABEL'),¨(⍕¨NT←(1=ST[;2])/ST[;3])),¨':')
[6]   NT_CODE←,(NT_CODE,[1.5](NT_ARG¨NT)),[2]⊂'→0'
[7]   Z←HEAD,NT_CODE,T_CODE
    ∇
```

The implementation of the function PI_COMP is straightforward. We know that the body of the function PI consists of one large case-statement which provides a clause for each symbol in the grammar. So, we have to generate the function header of PI and the branching code (line 2 of PI_COMP), the clauses for the terminal symbols (lines 3 and 4 of PI_COMP), and the clauses for the nonterminal symbols (lines 5 and 6 of PI_COMP).

```
    ∇Z←NT_ARG NT;B
[1]   Z←'Z←+/(⊂D-1)(YK MEMO ''YK'')¨'
[2]   →(1=+/(B←∊'(',¨(⍕¨PT[∊ST[NT;4];2]),¨')')∊'(')/PUSH
[3]   Z←Z,B
[4]   →0
```

```
[5]   PUSH:Z←Z,',⊂',B
    ∇
```

The code generator for a nonterminal symbol is shown in function NT_ARG which applies the function YK to the symbol string $y_1 \ldots y_k$ on the right hand side of a production rule. From the number of derivation steps d given as left argument and the sentential form $y_1 \ldots y_k$ as right argument, YK computes the number of words which can be generated with exactly d derivation steps starting from $y_1 \ldots y_k$. Of course, we have to apply YK to each possible right-hand side of the nonterminal and to take the sum over all possibilities.

By all ordered k-partitions of d derivation steps, we mean all k-tuples of integers $r_1, \ldots, r_k$ such that $r_1 + r_2 + \ldots + r_k = d$, with $r_i \geq 0$. For one k-partition of d the number of words is the product $\prod_{i=1}^{k} \Pi(y_i, r_i)$. In [Geyer-Schulz, 1995] an elegant, but highly inefficient implementation of the function YK for $\Gamma(y_1 \ldots y_k, d)$ has been presented which computes all products for all k-partitions of d. However, a less elegant, but more efficient algorithm for computing $\Gamma(y_1 \ldots y_k, d)$ is shown below. The strategy of this algorithm which is due to H. Hõrner relies on the elimination of all products $\prod_{i=1}^{k} \Pi(y_i, r_i)$ with at least one $\Pi(y_i, r_i) = 0$.

```
      ∇Z←D YK Y;M
[1]    →(0>D)/Z←0
[2]    Z←+/2⊃D YKF MEMO 'YKF' Y
      ∇
```

Function YKF starts by retrieving all search space sizes for words starting with nonterminals in $y_1 \ldots y_k$ for all derivation steps up to d (line 1). Next, in line 2 we generate a list of all combinations of derivation steps $j = 1, \ldots, d$ starting with y_i, $i = 1, \ldots, k$ with non-empty search space. In line 3 we select those combinations which are k-partitions of d and we test if k-partitions of d exist. In line 4 we compute the index structure for accessing the search space sizes in our vector list A and in line 5 we return the existing k-partitions and the products.

```
      ∇Z←D YKF Y;MAX;A;I
[1]    A←PI MEMO 'PI'¨((Y←,Y)∘.,0,⍳D)
[2]    Z←,⊃(∘.,)/((0<⊂[2]A)SEL¨⊂0,⍳D)
[3]    →(0=⍴Z←(D=+/¨Z)/Z)/END
[4]    I←⊂¨¨((⊂⍳1↑⍴A),¨¨1+Z)
[5]    Z←Z P←×/¨I PICK¨⊂A
[6]    →0
[7]   END:Z←(0⍴0)0
      ∇
```

The functions SEL encapsulates the APL primitive selection, the function PICK implements a scatter index function.

```
      ∇Z←A SEL B
[1]    Z←A/B
      ∇

      ∇Z←I PICK A
[1]    Z←I⊃¨⊂A
      ∇
```

B THE ALGORITHM

```
    ∇Z←INIT_UNIFORM A;W;D;S
[1]   (W D)←A
[2]   Z←INIT_WORD_U W(((?(¯1+2*31))÷2*31)IN+\S÷+/S←(PI MEMO 'PI')
      ¨W,¨⍳D)
    ∇
```

The right argument of the word generating function INIT_WORD_U consists of the start symbol and the number of derivations drawn according to (7).

```
    ∇Z←N IN F
[1]   Z←1↑(N≤F)/⍳⍴F
    ∇
```

The word generating function

The pseudocode of the word generating function INIT_WORD_U is shown in Figure 8.

```
    ∇Z←INIT_WORD_U A;S;D;W;N;I;R;DN
[1]   Z←1⊃(W D)←A
[2]   →(~ST[W;2])/0
[3]   S←PI_TF_OR(,⊃ST[W;4])(I←D-1)
[4]   R←0⍴DN←0,,⊃PI_TF_AND(1↓N←0,,⊃PT[1↑S;2])(I)
[5]   LOOP:→(0=⍴N←1↓N)/END
[6]   R←R,⊂INIT_WORD_U(1⊃N)(1⊃DN←1↓DN)
[7]   →LOOP
[8]   END:Z←(W R)
    ∇
```

The choice function for production rules

The probability of choosing the right hand side of a production rule is computed in lines 3 to 5 of PI_TF_OR according to (8).

```
    ∇Z←PI_TF_OR ARGS;SUM;PSUMS;D;S;P
[1]   (S D)←ARGS
[2]   →(1=⍴Z←S)/0
[3]   PSUMS←(0,⍳D)∘.(YK MEMO 'YK')(,¨⊃¨PT[S;2])
[4]   P←,(¯1,(1↓⍴PSUMS))↑(0≠+/[2]PSUMS)/[1]PSUMS
[5]   Z←S[(((?(¯1+2*31))÷2*31)IN+\P÷+/P]
    ∇
```

The choice function for *k*-partitions

The probability of choosing a partition is computed in lines 3 and 4 of PI_TF_AND according to (9).

```
    ∇Z←PI_TF_AND ARGS;M;Y;D;P
[1]   Z←2⊃(Y D)←ARGS
[2]   →(1=⍴Y)/0
[3]   (M P)←(¯1↑(0≠(⍳D)YK MEMO 'YK'¨⊂Y)/⍳D)YKF MEMO 'YKF' Y
[4]   Z←M[(((?(¯1+2*31))÷2*31)IN+\P÷+/P]
    ∇
```

C COUNTING FOR L_{XOR}

The function PI implements the recursive function $\Pi(y,d)$ shown in (6). This function has been automatically generated by the function PI_COMP of Appendix A. Lines 1 to 4 of PI contain the branching code for the case-statement. The four clauses of the nonterminal symbols of L_{XOR} are coded in lines 5 to 16 of PI. The clauses for the terminal symbols of L_{XOR} are shown in lines 17-25 of PI. In Definition 6 this is the `otherwise` case.

For readers who want to verify that PI really corresponds to the (mathematical) Definition 6, the symbol table used is shown in Figure 9.

```
     ∇Z←PI A;X;D
[1]    (X D)←A
[2]    ⍎'→LABEL',⍕X
[3]    Z←'ERROR'
[4]    →0
[5]    LABEL1:
[6]    Z←+/(⊂D-1)(YK MEMO 'YK')¨(5 2 6)(5 3 1 6)(5 4 1 1 6)
[7]    →0
[8]    LABEL2:
[9]    Z←+/(⊂D-1)(YK MEMO 'YK')¨(7)(8)
[10]   →0
[11]   LABEL3:
[12]   Z←+/(⊂D-1)(YK MEMO 'YK')¨(9)
[13]   →0
[14]   LABEL4:
[15]   Z←+/(⊂D-1)(YK MEMO 'YK')¨(10)(11)
[16]   →0
[17]   LABEL5:
[18]   LABEL6:
[19]   LABEL7:
[20]   LABEL8:
[21]   LABEL9:
[22]   LABEL10:
[23]   LABEL11:
[24]   Z←1×(D=0)
[25]   →0
     ∇
```

FOUNDATIONS OF GENETIC ALGORITHMS • 4

EDITED BY
RICHARD K. BELEW
AND
MICHAEL D. VOSE

ISSN 1081-6593
ISBN 1-55860-460-X
Artificial Intelligence
Mathematics
Biology

Morgan Kaufmann Publishers, Inc.
San Francisco, California