Two Fast Tree-Creation Algorithms for Genetic Programming

Sean Luke

Abstract— **Genetic programming is an evolutionary optimization method that produces functional programs to solve a given task. These programs commonly take the form of trees representing LISP s-expressions, and a typical evolutionary run produces a great many of these trees. For this reason, a good tree-generation algorithm is very important to genetic programming. This paper presents two new tree-generation algorithms for genetic programming and for "strongly-typed" genetic programming, a common variant. These algorithms are fast, allow the user to request specific tree sizes, and guarantee probabilities of certain nodes appearing in trees. The paper analyzes these two algorithms and compares them with traditional and recently proposed approaches.**

Keywords— **Genetic Programming, Population Initialization, Tree Creation, Subtree Mutation, Tree Growth, Introns, Bloat**

I. Introduction

GENETIC programming (GP) is a variant of the genetic algorithm which uses simulated evolution to discover functional programs to solve some task. The most common form of genetic programming (and the one considered in this paper) evolves programs as LISP-like "program-trees" of function-nodes [11]. These trees serve both as the genetic material and as the resultant program individual; there is no intermediate representation.

GP needs a good random tree-creation algorithm to create trees that form the initial population, and to create subtrees used in *subtree mutation*, a GP operator used for modifying trees in the population. For the fast-increasing number of GP experiments that use various forms of subtree mutation as their primary means of modifying individuals, tree creation is critical because the generation of 100,000 trees or more is not uncommon.

Tree-creation also plays an important role in the current debate over *tree bloat*, the tendency of GP trees to grow during the evolutionary process independent of any increase in fitness [19], [17], [2], [21], [12]. Bloat is blamed for slowing down the evolutionary process by making individuals more resistant to change, by slowing the overall speed of evaluation and breeding, and by increasing memory requirements. Much of this debate centers around specific tree-modification operators. Some have argued that *subtree crossover* (another traditional GP tree-modification operator) is the chief culprit of bloat, especially in comparison to subtree mutation [1]. Others instead indict subtree mutation [13], going so far as to argue that in such cases fitness-selection pressure also causes bloat. However, as shown later in this paper, there are examples where subtree mutation can provably increase average tree size, even without any selection pressure at all.

The traditional GP tree-creation algorithm, GROW in [11], has serious weaknesses. Most significantly, GROW offers no user control over expected tree size, except through an absolute maximum depth bound. As discussed later, left to its own devices, GROW will often generate infinite trees on average, depending on the function set. Also, most versions of GROW typically do not offer fine-grained control over the expected probability that a particular function will appear in a tree. Given the considerable interest in introns, internode dependencies and semantics, and other issues of node interactions [6], [18], [8], [14], the ability to rigorously control specific node appearance is timely.

This paper discusses GROW, its variants, and recent proposed alternatives. It then offers two new algorithms that attempt to address the deficiencies in previous algorithms. These two new algorithms let the user request either an average tree size or specify a distribution of tree sizes from which to generate trees. Unlike recent alternatives, these two algorithms do not promise uniformly random tree structures. Instead they guarantee user-specified probabilities of occurrence for specific terminal and nonterminal functions within the generated trees. Lastly, unlike recent alternatives, these two algorithms are fast, running in near-linear time, or in linear time under reasonable constraints. The paper analyzes the new algorithms, then gives additional versions of the algorithms tailored for "strongly-typed" genetic programming [16].

II. Definitions

This paper uses certain critical symbols in the description and analysis of various constructive tree-generation algorithms.

$\mathbf{T}$ denotes a newly generated genetic program tree. S is the maximal number of nodes in a tree permitted by a given tree-generation algorithm, and D is the maximal depth of a tree permitted by the algorithm. E_{tree} is the expected tree size of $\mathbf{T}$. $\mathbf{T}$ is created by choosing nodes from a function set F, divided two disjoint sets, *terminals* T (leaf nodes), and *nonterminals* N (interior nodes).

p is the probability that, in the process of choosing nodes to form $\mathbf{T}$, a given tree-creation algorithm will pick a nonterminal from the function set (as opposed to a terminal). b is the expected number of children to nonterminal nodes picked from the function set. g is the expected number of children to a newly generated node in $\mathbf{T}$. It is important to note that since the expected number of children of a terminal is 0, and the expected number of children to a nonterminal is p, then $g = pb + (1-p)(0) = pb$.

III. The Traditional Tree-Creation Algorithm

By far the most common mechanism for creating trees and subtrees in GP is the GROW tree-creation algorithm [11]. GROW and a full-tree variant (FULL) are used to generate trees for the initial population at the beginning of the evolutionary process. GROW is also used almost universally to generate new subtrees for "point" subtree mutation and is very popular for other special kinds of subtree mutation. For some examples of point mutation and other mutation approaches, see [11], [1], [4], [5], [14], [15],

Domain	b	p	E_{tree}
Cart Centering	$\frac{11}{6}$	$\frac{3}{4}$	∞
Ant	$\frac{7}{3}$	$\frac{1}{2}$	∞
Regression	$\frac{3}{2}$	$\frac{8}{9}$	∞
11-Multiplexer	2	$\frac{4}{15}$	$\frac{15}{7}$
6-Multiplexer	2	$\frac{2}{5}$	5
3-Multiplexer	$\frac{5}{3}$	$\frac{1}{2}$	6

Table 1. Algorithmic results for the introductory domains from [11] chapter 7, using the GROW algorithm. b is the expected number of children of a nonterminal picked from the function set. p is the probability of choosing a nonterminal at tree-creation time. E_{tree} is the expected size of a tree. From Theorem 1, this is $\frac{1}{1-pb}$ if $pb < 1$, ∞ otherwise.

[13]. Though little studied, GROW and FULL appear in the lion's share of existent GP literature and are the chief or only tree-creation algorithms for nearly all popular GP libraries (including lil-gp, GPSys, GPQuick/GPData, GPC++, DGPC, SGPC, and Koza's Simple-LISP code [11]).

GROW begins with a set of functions F to place as nodes in the tree. Each nonterminal function in the set has a specific arity. This algorithm randomly selects a root from the full set of functions (both terminals and nonterminals), then fills the root's arguments with random functions, then *their* arguments with random functions, and so on. A common variant is shown below:

```
Given:
   maximum depth bound D
   function set F consisting of nonterminal set N and terminal set T
Do:
   New tree T = GROW(0)

GROW(depth d)
   Returns: a tree of depth ≤D − d

   If d = D, return a random terminal from T
   Else
      Choose a random function f from F                    ○
      If f is a terminal, return f
      Else
         For each argument a of f,
            Fill a with GROW(d + 1)
         Return f with filled arguments
```

GROW's companion algorithm (FULL) is used to force the generation of full trees. It differs from GROW only in the line marked ○, where f is chosen at random from N, not F. That is, f is always a nonterminal. If N is nonempty, then FULL always produces a full tree "trimmed' to depth D, and so produces a very narrow range of tree structures, with relatively less applicability than GROW. This paper focuses primarily on the dynamics of GROW.

Some variants of GROW permit the user to specify a maximum tree size S. They enforce this either by producing trees repeatedly until one of size less than S is created, or by keeping track of the number of created nodes (so far) plus the number of unfilled arguments; when this total exceeds S, only terminals may fill arguments from then on (see [5] for an interesting example).

While GROW is easy to implement and runs in linear time, it has three weaknesses. First, the algorithm picks all functions with equal likelihood; there is no way to fine tune the preference of certain functions over others. Second, the algorithm does not give the user much control over the tree structures generated. Third, and most significant, while D (or S) is used as an upper bound on maximal tree depth, there is no appropriate way to create trees with either a fixed or average tree size or depth.

The lack of a rigorous way to specify tree size is very problematic: in most GP literature, the sets of nonterminals and terminals are picked based on domain need, with little consideration given to their effect on tree generation. The expected tree size E_{tree} under the traditional algorithm is determined solely by g, the expected number of children of a newly generated node. In particular, if $g < 1$, then $E_{tree} = \frac{1}{1-g}$, else E_{tree} is infinite, as shown in Theorem 1.

Lemma 1: Assume an algorithm that "grows" a tree from a randomly chosen root node by attaching randomly generated child nodes into unfilled argument positions of nodes currently in the tree. This is done for example in GROW. Let $g \geq 0$ be the expected number of children of a newly generated node. Then at depth d in the tree, the expected number of nodes is $E_d = g^d$.

Proof: The expected number of nodes E_d at depth d is

$$E_d = \begin{cases} 1 & \text{if } d = 0 \\ E_{d-1} g & \text{if } d > 0 \end{cases}$$

From this it follows that $E_d = g^d$. ■

Theorem 1: As in Lemma 1, assume an algorithm which "grows" a tree from a randomly chosen root node by attaching randomly generated child nodes into unfilled argument positions of nodes currently in the tree. This is done for example in GROW. Let E_{tree} be the expected size of a tree built with the assumed algorithm. Let $g \geq 0$ be the expected number of children of a node newly generated by the algorithm. If and only if $g < 1$, then E_{tree} is finite ($E_{tree} = \frac{1}{1-g}$), else it is infinite.

Proof: The expected size of a tree is the sum of the expected number of nodes at all levels, that is, $E_{tree} = \sum_{d=0}^{\infty} E_d$. From Lemma 1, $E_{tree} = \sum_{d=0}^{\infty} g^d$. From the geometric series, for $g \geq 0$,

$$\sum_{d=0}^{\infty} g^d = \begin{cases} \frac{1}{1-g} & \text{if } g < 1 \\ \infty & \text{if } g \geq 1 \end{cases}$$

■

For this reason, poor function set choices can have a dramatic unforeseen effect on tree creation. Consider the following example: imagine a typical domain that has 5 terminals and 5 nonterminals, where the average number of children of a nonterminal is 2. In this case, $g = 1$, and so the expected tree size is infinite! Although the complexity of GROW is linear in the size of the tree, this doesn't say much in the face of infinite tree sizes. As such, the worst-case time bound for GROW is in fact dependent entirely on the choice of functions in the function set.

Tweaking the function set to come up with a combination of terminals and nonterminals that give a reasonable E_{tree} is often difficult; very slight modifications in a function set can result in an E_{tree} that is either very small (say, less than 2) or infinite. As a result, many common published function sets inadvertently

have either very small or very large, possibly infinite, expected tree sizes. For example, Table 1 shows that three of the four introductory examples in [11] have an infinite expected tree size (cart centering, regression, ant). The fourth (11-multiplexer) has an average tree size of about 2.

The classic genetic programming code described in [11] uses several ad-hoc methods to compensate for the tree size resulting from GROW. First, it imposes a small maximal depth D (from 2 to 6) on generated trees. Oddly, the maximal depth is not chosen at random from this range, but in a round-robin fashion. Because GROW so often creates infinite-sized trees, this maximal depth limit "shaves" function trees to keep the initial population size reasonable, resulting in an excessive number of full or near-full trees. Second, it rejects all trees of depth 1, and eliminates duplicate trees, which increases average tree size. Third, when creating initial trees, a mixture of GROW and FULL is used, but when creating subtrees for modifying trees through "point" mutation, only GROW is used. Newly mutated trees are rejected if they exceed an absolute depth limit (typically 17).

IV. Previous GP Tree-Creation Algorithms

Previous improvements to GROW and FULL have focused on generating uniformly random tree structures of predetermined sizes.

Iba's RAND_tree algorithm [10] generates uniform tree structures by using Dyck words to build trees bottom-up. RAND_tree builds trees from a fixed-size pool of tree nodes, joining nodes together to form subtrees, and ultimately joining subtrees together to form the final tree. RAND_tree makes certain that each node in the tree has an arity selected from a user-supplied arity set (for example, all nonterminals might have either 2, 3, or 5 children). This arity constraint puts RAND_tree in conflict with more restrictive forms of GP (such as "strongly-typed" GP [16], where each function has a specific return type and distinct argument types). To use strongly-typed GP with RAND_tree, the user must create a function set with all permutations of both the arity set and return types, else the algorithm will generate invalid tree structures.

Other approaches have tried production grammars [20], [7]. Böhm and Geyer-Schulz [3] extend this approach by selecting trees with exact uniform probability from a tree-derivation grammar. Given the absolute maximum bound on tree size S, their approach first compiles (off-line) a table $\Pi(W, s)$ of probabilities of producing trees of size $s \leq S$ derived from some symbol W. Once this table has been compiled, their tree-generation algorithm first picks a statistically random tree size and start symbol. It then expands this symbol with some random production, using the table to recursively compute appropriate sizes for each subtree that will be derived from the symbols in the expansion. This elegant approach can generate traditional GP or strongly-typed GP trees of any size (up to S) from a completely uniform random distribution of tree structures.

The strength of all these approaches is that they permit user control over the size of the trees generated, and generate uniformly-distributed random tree structures. But there are two drawbacks to these approaches: they are combinatorically very slow, and they cannot guarantee user-defined probabilities of appearance of functions within their trees (because this conflicts with generating uniformly-distributed structures).

Iba notes that RAND_tree has very high (in some cases infinite) computational complexity because the tree-structure determination includes producing large Catalan numbers. Böhm and Geyer-Schulz's algorithm has linear complexity once the table Π has been compiled, but compiling this table includes effectively enumerating all possible appropriate trees of size $\leq S$. Even with the help of dynamic programming, the complexity of this generation can be very high, though possibly polynomial. Böhm and Geyer-Schulz do not give a worst-case bound for generating this table. The authors note that combinatorics and other issues could make the practical application of the algorithm difficult.

V. PTC1 and PTC2

This paper offers two alternative tree-creation algorithms, Probabilistic Tree-Creation (PTC) 1 and 2, which take a different approach from past algorithms. Like past algorithms, PTC1 and PTC2 give the user control over generated tree size. However, these new algorithms do not attempt to generate completely uniformly distributed tree structures. Instead, they guarantee what previous approaches cannot: user-defined probabilities of appearance of functions within the tree. But most importantly, PTC1 and PTC2 have very low computational complexity (linear, under reasonable constraints).

PTC1 is a modification of GROW that allows the user to provide probabilities of appearance of functions in the tree, plus a desired *expected tree size*, and guarantees that, on average, trees will be of that size. PTC1 has formal results that have applicability to GROW. However, PTC1 does not give the user any control over the variance in tree sizes generated, which limits its usefulness.

With PTC2, the user provides a probability distribution of requested tree sizes. PTC2 guarantees that, once it has picked a random tree size from this distribution, it will generate and return a tree of that size or slightly larger. This approximation is less precise than PTC1, and PTC2 does not yield the same elegant theoretical results. However, it gives the user real control over tree size variance, a critical advantage.

The remainder of this paper discusses PTC1 and PTC2, giving an analysis and complexity results. It then gives similar algorithms for strongly-typed genetic programming.

VI. The PTC1 Algorithm

The PTC1 algorithm is as follows: the set of functions F is divided into two disjoint subsets: nonterminals N and terminals T. During tree-generation time, the algorithm will alternately choose new child nodes from either the nonterminals or from the terminals. For each nonterminal n in N the user provides a probability q_n that n will be chosen from N when the algorithm needs a nonterminal. Similarly, for each terminal t in T the user provides a probability q_t that t will be chosen from T when the algorithm needs a terminal. The user also provides a maximum depth bound D as before, though this bound is used only to enforce an absolute, and preferably large, memory restriction. Lastly, the user indicates an expected tree size E_{tree}.

Before generating any trees, the algorithm computes p, the probability of choosing a nonterminal over a terminal in order to produce a tree with an expected tree size E_{tree}, as

$$p = \frac{1 - \frac{1}{E_{tree}}}{\sum_{n \in N} q_n b_n}$$

where b_n is the arity of nonterminal n. This computation need be done only once offline. Then the algorithm proceeds to create the tree:

Given:
maximum depth bound D
function set F consisting of nonterminal set N and terminal set T
computed probability of choosing a nonterminal p
probabilities q_t and q_n for each $t \in T$ and $n \in N$
Do:
new tree **T** = PTC1(0)

PTC1(depth d)
Returns: a tree of depth $\leq D - d$

If $d = D$, return a terminal from T (by q_t probabilities)
Else if, with probability p a nonterminal must be picked,
Choose a nonterminal n from N (by q_n probabilities)
For each argument a of n,
Fill a with PTC1($d + 1$)
Return n with filled arguments
Else return a terminal from T (by q_t probabilities)

This algorithm guarantees an expected tree size of E_{tree} for trees and subtrees by determining the appropriate nonterminal-selection probability p. In the trivial case where there are *only* terminals in the function set, the algorithm of course cannot provide any E_{tree} other than 1. Additionally, by adjusting nonterminals with large fan-outs to have a lower (or higher) probability of occurrence than nonterminals with small fan-outs, the user can bias the typical "bushiness" of a tree, yet keep E_{tree} the same.

PTC1 attempts to fix the expected tree size E_{tree} yet still provide the user with as much freedom as possible in defining probabilities of appearance for each function. Recognize that E_{tree} can be controlled by fixing g, the expected number of children of a newly generated node, shown in Theorem 2 below. Note that since $g = \sum_{f \in F} q_f b_f$, to fix g over some user-defined set F of functions with known arities (b_f), the algorithm must somehow adjust the relative appearance (q_f) of functions within the set. PTC1 accomplishes this simply by picking the right p, the probability that, at node-creation time, a node will be picked from the nonterminal set (as opposed to the terminal set), as shown in Theorem 3 below. Within the respective nonterminal or terminal sets, the user is still free to set his own q_t and q_n.

Theorem 2: For PTC1, assume that N, the set of nonterminal functions, is nonempty. Let p be the probability that a newly generated node will be a nonterminal. Let b be the expected number of children of a nonterminal node picked from the function set. Let g be the expected number of children of a newly generated node. Then a p can be predetermined to guarantee any specific $E_{tree} \geq 0$.

Proof: Since N is nonempty, therefore $b > 0$. Since $g = pb$, given a constant, nonzero b, a p can clearly be picked to produce any desired g. From Theorem 1, a g (and hence a p) can thus be picked to determine any finite or infinite $E_{tree} \geq 0$. ■

Theorem 3: For PTC1, assume that N, the set of nonterminal functions, is nonempty. For each $n \in N$, let b_n be the arity of n, and let q_n be the probability that a newly generated nonterminal will be n. Let p be the probability that a newly generated node will be a nonterminal. Let b be the expected number of children of a nonterminal node picked from the function set, that is, $b = \sum_{n \in N} q_n b_n$. Then the nonterminal-choice probability p necessary to guarantee that a tree **T** will be of expected finite size $E_{tree} > 0$ is $p = \frac{1 - \frac{1}{E_{tree}}}{\sum_{n \in N} q_n b_n}$.

Proof: As a consequence of Theorems 1 and 2, $E_{tree} = \frac{1}{1 - pb}$. Since N is nonempty, $b > 0$, hence p may be determined as $p = \frac{1 - \frac{1}{E_{tree}}}{b}$. Replacing b with $\sum_{n \in N} q_n b_n$ yields $p = \frac{1 - \frac{1}{E_{tree}}}{\sum_{n \in N} q_n b_n}$. ■

VII. Complexity of PTC1

The time bound for PTC1 is determined by the complexity c_T of choosing a random terminal from some probability distribution of terminals, and the complexity c_N of choosing a random nonterminal from a probability distribution of nonterminals. From Theorem 4 below, the number of nonterminals in a tree is pE_{tree} and the expected number of terminals is $(1 - p)E_{tree}$, hence the complexity of generating a full tree of terminals and nonterminals is $O(c_N pE_{tree} + c_T(1 - p)E_{tree})$.

Theorem 4: For PTC1, let E_{tree} be the expected tree size, and p be the precomputed nonterminal-selection probability to generate a tree of expected size E_{tree}. Then the expected number of nonterminals in a tree is $E_{n,tree} = pE_{tree}$ and the expected number of terminals is $E_{t,tree} = (1 - p)E_{tree}$.

Proof: Let g be the expected number of children of a newly generated node under PTC1. Then the expected number of nonterminals $E_{n,d}$ at depth d is

$$E_{n,d} = \begin{cases} p & \text{if } d = 0 \\ E_{n,d-1} bp & \text{if } d > 0 \end{cases}$$

That is, $E_{n,d}$ is equal to the expected number of nonterminals $E_{n,d-1}$ at depth $d - 1$, times the expected number of children b to each of these nonterminals, times the probability p that these children are nonterminals themselves.

From this it follows that $E_{n,d} = p(bp)^d = pg^d$. Since by Lemma 1, the expected number of nodes $E_d = g^d$, then the number of terminals is $E_{t,d} = E_d - E_{n,d} = g^d - pg^d = (1 - p)g^d$. Hence for the whole tree, the expected number of nonterminals in a tree is then $E_{n,tree} = \sum_{d=0}^{\infty} E_{n,d} = p \sum_{d=0}^{\infty} g^d$ and the expected number of terminals is $E_{t,tree} = \sum_{d=0}^{\infty} E_{t,d} = (1 - p) \sum_{d=0}^{\infty} g^d$. From Theorem 1 we get $E_{n,tree} = pE_{tree}$ and $E_{t,tree} = (1 - p)E_{tree}$. ■

Note that c_N or c_T is at most the complexity of a binary search through some probability distribution. To achieve this for the nonterminal probabilities, for example, arrange all the q_n into disjoint intervals from 0 to 1 corresponding to each $n \in N$:

$$\begin{array}{ll} n_1: & [0\ ,\ q_{n_1}) \\ n_2: & [q_{n_1}\ ,\ q_{n_1} + q_{n_2}) \\ n_3: & [q_{n_1} + q_{n_2}\ ,\ q_{n_1} + q_{n_2} + q_{n_3}) \\ \ldots & \ldots \\ n_{|N|}: & [1 - q_{n_{|N|}}\ ,\ 1] \end{array}$$

A random event points somewhere into this set of intervals; an $O(\lg(\|N\|))$ binary search through this set finds which n cor-

Domain	Natural E_{tree}	# Size 1 Trees when E_{tree} is... Natural	4	16	256
Cart Centering	∞	$\frac{1}{4}$	$\frac{13}{22}$	$\frac{43}{88}$	$\frac{643}{1408}$
Ant	∞	$\frac{1}{2}$	$\frac{19}{28}$	$\frac{67}{112}$	$\frac{1027}{1792}$
Regression	∞	$\frac{1}{9}$	$\frac{1}{2}$	$\frac{3}{8}$	$\frac{43}{128}$
11-Multiplexer	$\frac{15}{7}$	$\frac{11}{15}$	$\frac{5}{8}$	$\frac{17}{32}$	$\frac{257}{512}$
6-Multiplexer	5	$\frac{3}{5}$	$\frac{5}{8}$	$\frac{17}{32}$	$\frac{257}{512}$
3-Multiplexer	6	$\frac{1}{2}$	$\frac{11}{20}$	$\frac{7}{16}$	$\frac{103}{256}$

Table 2. Expected number of size-1 trees generated, as a percentage of the whole population. "Natural" indicates numbers if tree generation is made without restrictions on tree size (as in GROW). Other columns give numbers when PTC1 restricts the expected tree size E_{tree} to various values. The probability of generating a size-1 tree is the probability of generating a terminal (e.g., $1-p$).

responds to the random event. This can be done similarly for terminal probabilities (q_t), hence averaged over several iterations, an upper complexity bound on PTC1 is

$$O(\lg(\|N\|)pE_{tree} + \lg(\|T\|)(1-p)E_{tree}) \leq O(\lg(\|F\|)E_{tree})$$

If c_T and c_N were constant, then the complexity would reduce to $O(E_{tree})$. This might happen if all the terminals and nonterminals had equal q probabilities, in which case selecting a random terminal or nonterminal can be done with a simple $O(1)$ random event as in GROW. An $O(E_{tree})$ complexity can also be achieved if the q_t and q_n probabilities are discrete values. For example, imagine that there were three nonterminals with probabilities $\{q_1 = .2, q_2 = .3, q_3 = .5\}$. One can create an array $[n_1, n_1, n_2, n_2, n_2, n_3, n_3, n_3, n_3, n_3]$. At nonterminal-selection time, picking randomly from this array is $O(1)$.

VIII. PTC2

PTC1 generates trees expected around a specific user-defined tree size. A serious problem with PTC1 is that it does not give the user control over *variance* in tree size. PTC1, like GROW, produces a large number of small trees; there is little the user can do about it. For example, consider the large number of trees of size 1 (equal to $1-p$) generated under the previous example (five nonterminals and five terminals, and an average nonterminal arity of 2). Using PTC1 enforcing an expected tree size of 10, about 11/20 of all new trees would be of size 1. Similarly, under GROW (no enforcement), exactly half of the trees generated would be of size 1, even though the expected tree size is infinity!

In general, when E_{tree} is restricted to be less than GROW's expected tree size, then PTC1 generates more trees of size 1 than GROW would. If the enforced expected tree size is larger GROW's expected tree size, PTC1 will generate fewer small (or size 1) trees than GROW. Table 2 illustrates this for the introductory domains from [11].

PTC2 avoids this problem by allowing the user to provide beforehand a probability distribution of requested tree sizes. Like PTC1, PTC2 also guarantees user-provided distributions of nonterminals and terminals appearing in each tree. And like PTC1, PTC2 is very fast. However, PTC2 is not as elegant as PTC1: when it picks a tree size from the distribution, it may produce a tree of that size or slightly larger. In effect, while PTC2 *guarantees* the user-provided nonterminal and terminal probabilities of appearance, it *approximates* the user-provided tree-size distribution.

PTC2 is as follows: the set of functions F is divided into two disjoint subsets: nonterminals N and terminals T. For each nonterminal n in N the user provides a probability q_n that n will be chosen from N when the algorithm needs a nonterminal. Similarly, for each terminal t in T the user provides a probability q_t that t will be chosen from T when the algorithm needs a terminal. The user also provides a maximum depth bound D, a maximum size bound S, and a probability distribution of desired tree sizes $w_1, ..., w_S$ for each tree size from 1 to S.

Given:
- maximum depth bound D
- maximum size bound S
- function set F consisting of nonterminal set N and terminal set T
- probabilities q_t and q_n for each $t \in T$ and $n \in N$
- probabilities $w_1, ..., w_S$ of generating a tree of size $\{1, ..., S\}$

Do:
- new tree **T** = PTC2()

PTC2()
- *Returns:* a tree of depth $\leq D$ and a size between 1 and $S + b_{max}$ inclusive, where b_{max} is the largest number of arguments to any nonterminal in N
- Let r be a random tree size from the probability distribution $w_1, ..., w_S$
- If $r = 1$ return a random terminal from T (by q_t probabilities)
- Else
  - Choose a nonterminal n from N (by q_n probabilities)
  - Tree root $root \leftarrow n$
  - Node depth $d \leftarrow 1$
  - For each argument position a of n, Enqueue($\{a, d\}$)
  - Current tree size $s \leftarrow 1$
  - Until Size()$+s \geq r$ or Size()$= 0$,
    - {argument position a, depth d}$\leftarrow$RandomDequeue()
    - If $d = D$, Fill a with a terminal t from T (by q_t probabilities)
    - Else
      - Choose a nonterminal n from N (by q_n probabilities)
      - Fill a with n
      - For each argument position a of n, Enqueue($\{a, d+1\}$)
    - $s \leftarrow s + 1$
  - Until Size()$= 0$,
    - {argument position a, depth d}$\leftarrow$RandomDequeue()
    - Choose a terminal t from T (by q_t probabilities)
    - Fill a with t
  - Return $root$

The algorithm begins by picking a random tree size from the the user-provided tree-size probability distribution. It then attempts to build a tree of that size or slightly greater. The algorithm builds the tree by starting with a single node, $root$, extending the tree with nonterminals at random places along the current tree boundary. It continues this until the size of the unfilled positions along the boundary plus the number of nonterminals currently in the tree is greater than or equal to the requested size. Then the algorithm fills the remaining boundary positions with terminals, and returns the result.

Whenever the boundary extends beyond depth D, the offend-

ing boundary positions are automatically filled with terminals; this means that if D is so small that a full tree of S nodes might be greater than depth D, then PTC2 may return a tree smaller than expected.

To maintain the tree boundary, the algorithm stores in a global random queue the position and depth of each unfilled argument along the boundary, picking random items from this queue as needed. To do this, the algorithm relies on three random-queue functions: Size, which returns the size of the queue, Enqueue, which enqueues an item, and RandomDequeue, which dequeues and returns a random item. As shown below, the random queue can be implemented so that all three functions run in $O(1)$ time.

Given:
array of slots $U = \{u_0, ..., u_{S+b_{max}-1}\}$
array-fill value $h \leftarrow 0$

Size()
Returns: the size of the queue

return h

Enqueue(item i)
Returns: nothing

$u_h \leftarrow i$
$h \leftarrow h + 1$

RandomDequeue()
Returns: the value of a random slot in U, or nil if U is empty

If $h = 0$, return nil
Else
$h \leftarrow h - 1$
Let r be a random integer, $0 \leq r \leq h$
Swap the values of u_h and u_r
Return u_r

The random-queue implementation relies in this case on a maximum queue value of $S + b_{max}$, the largest returnable tree size. It can be instead implemented with a small initial queue array, extended when needed by doubling its size. This also yields an amortized complexity of $O(1)$ for all three operations.

IX. Complexity of PTC2

Because random-queue operations can be done in $O(1)$ time, and either a nonterminal or terminal is chosen at each iteration, the complexity of building a tree of requested size r is the time it takes to pick a random terminal or nonterminal (from the q_t and q_n distributions) multiplied by the number of iterations.

The first until-loop in PTC2 runs until Size()$+s \geq r$ or Size()$= 0$. In the first case, consider the last iteration of the first until-loop. As this iteration starts, Size()$+s < r$. The iteration may perform one last Enqueue before the iteration ends. Since the largest number of arguments to a nonterminal in N is b_{max}, this last enqueing operation will increase Size()$+s$ to no more than $r + b_{max}$. At the point between the two until-loops, the first loop has run for exactly $s - 1$ iterations, and the second loop will run for exactly Size() iterations. Hence the total number of iterations is $O(r + b_{max})$.

In the exceptional second case (which will only occur when D is inappropriately small relative to S), the first until-loop runs for no more than r iterations, else the first case would have been triggered. The second until-loop will then run for 0 iterations, hence the total number of iterations is $O(r) \leq O(r + b_{max})$.

As discussed, the complexity of choosing a nonterminal from N or a terminal from T is $O(lg(||N||))$ and $O(lg(||T||))$ respectively, or both $O(1)$ under reasonable constraints. Since at each iteration either a nonterminal or a terminal is chosen, a loose complexity bound for choosing nonterminals and terminals in the algorithm is $O((r + b_{max}) \times \max(lg(||N||), lg(||T||)))$, or $O(r + b_{max})$ under reasonable constraints.

Likewise, picking randomly from the tree-size probability distribution takes at most $O(lg(||S||))$ time, or $O(1)$ under reasonable constraints. Let r_{mean} be the mean tree size given the provided probability distribution. Then PTC2 has an average complexity of

$$O((r_{mean} + b_{max}) \times \max(lg(||N||), lg(||T||)$$

Under reasonable constraints as discussed earlier, this reduces to $O(r_{mean} + b_{max})$. Since the largest possible tree is of size $S + b_{max}$, the worst-case complexity is therefore

$$O((S + b_{max}) \times \max(lg(||N||), lg(||T||)))$$

which under reasonable constraints reduces to $O(S + b_{max})$, effectively linear. If D is too small relative to S and the exceptional second case is triggered, then the complexity may be even lower.

X. A Strongly-Typed PTC1 Algorithm

Under relaxed constraints, PTC1 can be extended easily to handle the "basic" form of strongly-typed genetic programming (STGP) [16]. Under STGP, types are associated with each argument and the return value of each function; at tree-creation time, a parent function may have a particular child in an argument position only if the parent's argument type (for that argument position) and the child's return type match. StronglyTypedPTC1 assumes that for each type, there exists at least one nonterminal and at least one terminal whose return values are of that type.

The algorithm presented is for the more common "basic" STGP without generic functions, as detailed in [16]. In order to accommodate STGP, StronglyTypedPTC1 must place further constraints on user-specified probabilities, by dividing the set of functions F into not just terminals and nonterminals, but also further subdividing these subsets by the functions' return types.

The algorithm is as follows: Let Y be the set of types. The set of functions F is divided into two disjoint subsets nonterminals N and terminals T. These subsets are further divided by their return types into subsets $Ny_1, Ny_2, \ldots Ny_z$, one for each $y \in Y$, and $Ty_1, Ty_2, \ldots Ty_z$, one for each $y \in Y$. During tree-generation time, the algorithm will, for some y, alternately choose new child nodes from either that N_y or T_y. For each nonterminal $n_y \in N_y$ the user provides a probability $q_{n,y}$ that n_y will be chosen when the algorithm needs a nonterminal with return type y. Similarly, for each terminal $t_y \in T_y$ the user provides a probability $q_{t,y}$ that t_y will be chosen from T_y when the algorithm needs a terminal with return type y. The user also provides a return type y_r for the tree, and a maximum depth bound D, though this bound is used only to enforce an absolute, and preferably large, memory restriction. Lastly, the user indicates an expected tree size E_{tree}.

Before generating any trees, the algorithm computes p_y for each $y \in Y$. p_y is the probability of choosing a nonterminal over a terminal of return type y in order to produce a tree with an expected tree size E_{tree}:

$$p_y = \frac{1 - \frac{1}{E_{tree}}}{\sum_{n_y \in N_y} q_{n,y} b_{n,y}}$$

where $b_{n,y}$ is the number of arguments for nonterminal n_y. This computation need be done only once at function-creation time.

Then the algorithm proceeds to create the tree:

Given:
maximum depth bound D
disjoint nonterminal subsets N_y of nonterminal set N for each $y \in Y$
disjoint terminal subsets T_y of terminal set T for each $y \in Y$
computed nonterminal-choice-probabilities p_y for each $y \in Y$
for each T_y and N_y,
probabilities $q_{n,y}$ and $q_{t,y}$ for each $t_y \in T_y$ and $n_y \in N_y$
return type for the tree $y_r \in Y$
Do:
new tree **T** = StronglyTypedPTC1(0,y_r)

StronglyTypedPTC1(depth d, return type $y \in Y$)
Returns: a tree of depth $\leq D - d$ and of return type y

If $d = D$, return a terminal from T_y (by $q_{t,y}$ probabilities)
Else if, with probability p_y a nonterminal must be picked,
Choose a nonterminal n_y from N_y (by $q_{n,y}$ probabilities)
For each argument a of n_y of argument type y_a
Fill a with StronglyTypedPTC1($d + 1$,y_a)
Return the completed nonterminal n_y with filled arguments
Else return a terminal from T_y (by $q_{t,y}$ probabilities)

Because of the user-provided type constraints of strongly-typed genetic programming, this version of PTC1 cannot guarantee that each terminal t will appear in the tree with some probability q_t relative to other terminals (or likewise a nonterminal n appearing with probability q_n relative to other nonterminals). Instead, it makes sure that each terminal t_y of a type $y \in Y$ will appear with probability $q_{t,y}$ relative to other terminals of type y, and similarly that each nonterminal n_y of type $y \in Y$ will appear with $q_{t,y}$ relative to other nonterminals of type y.

This algorithm guarantees an expected tree size of E_{tree} for all STGP trees by determining the necessary probability p_y for each return type y such that subtrees returning that return type will each be of E_{tree} size. Theorem 5 shows that the algorithm's method of picking of each p_y is correct and invariant over of y.

Theorem 5: Let Y be is the set of STGP return types. Let the set of nonterminals N be divided into nonempty subsets $N_{y_1}, N_{y_2}, \ldots N_{y_z}$, one for each $y \in Y$. At tree-building time, let p_y be the probability that a nonterminal will be chosen as a new child node for a particular type $y \in Y$. Given some y, for each $n_y \in N_y$, let $q_{n,y}$ be the probability that n_y will be chosen given that a nonterminal is to be chosen, and $b_{n,y}$ be the number of arguments to n_y. Let b_y be the expected number of children of a nonterminal node of return type y in the tree, that is, $b_y = \sum_{n_y \in N_y} q_{n,y} b_{n,y}$. Then under StronglyTypedPTC1, the nonterminal-choice probability p_y necessary to guarantee that a tree or subtree **T** of return type $y \in Y$ will be of expected finite size $E_{tree} > 0$ is: $p_y = \frac{1 - \frac{1}{E_{tree}}}{\sum_{n_y \in N_y} q_{n,y} b_{n,y}}$

Proof: Let $g \geq 0$ be the expected number of children of a node newly generated by the algorithm. From Theorem 1, $E_{tree} = \frac{1}{1-g}$ if and only if $g < 1$. Thus g may be determined from E_{tree} as $g = 1 - \frac{1}{E_{tree}}$.

For any $y \in Y$, since p_y denotes the likelihood that the newly generated node function will be a nonterminal, and the expected number of children to a terminal is 0, then the expected number of children of the newly generated node is $g = p_y b_y + (1 - p_y)(0) = p_y b_y$. Therefore $p_y = \frac{1 - \frac{1}{E_{tree}}}{b_y}$.

Since N_y is nonempty, $b_y > 0$, so for any given b_y and requested E_{tree}, an appropriate p_y may always be determined. Replacing b_y with $\sum_{n_y \in N_y} q_{n,y} b_{n,y}$ yields $p_y = \frac{1 - \frac{1}{E_{tree}}}{\sum_{n_y \in N_y} q_{n,y} b_{n,y}}$ ■

The algorithm can be modified to permit types for which there exists no nonterminal with a return value for that type; for each such type y, simply set p_y to 0. However, this does not guarantee that the expected size of the tree will remain E_{tree}, only that it will be no larger than E_{tree}.

The complexity of this algorithm is dependent largely on the size of each set of functions by type, and the combinations of types of arguments to each function. However, the complexity is no worse than the complexity for PTC1 under ordinary GP (that is, ignoring the types to the functions in question). This is because the number of nonterminals that must be searched is no more than $\|N\|$, and the number of terminals is no more than $\|T\|$.

The algorithm can also be modified to accommodate a more advanced STGP with "general functions" (functions with more than one valid return type) [16], [9]. In this case the various sets N_{y_x} are not required to be disjoint, nor are the various sets T_{y_x}. Choosing a nonterminal from some N_{y_x} is still no more difficult than $O(\|N\|)$ and choosing a nonterminal from some T_{y_x} is no more difficult than $O(\|T\|)$. However, since $\|Y\|$ may now be greater than $\|N\|$ and $\|T\|$; the complexity of PTC1 with general functions is bounded by the maximum of the PTC1 complexity bound and $\|Y\|$. Of course $\|Y\|$ may be any size the designer likes, but it is rarely larger than $\|F\|$; at any rate, $\|Y\| \leq 2^{\|F\|}$, else Y will contain duplicate types.

XI. A Strongly-Typed PTC2 Algorithm

PTC2 can also be extended to handle strongly-typed genetic programming, assuming relaxed constraints similar to StronglyTypedPTC1. And just like StronglyTypedPTC1, StronglyTypedPTC2 assumes that for each user-provided type, there exists at least one nonterminal and at least one terminal whose return values are of that type.

The algorithm works as follows: the set of functions F is divided into two disjoint subsets: nonterminals N and terminals T. These subsets are further divided by their return types into subsets $Ny_1, Ny_2, \ldots Ny_z$, one for each $y \in Y$, and $Ty_1, Ty_2, \ldots Ty_z$, one for each $y \in Y$. During tree-generation time, the algorithm will, for some y, alternately choose new child nodes from either that N_y or T_y. For each nonterminal $n_y \in N_y$ the user provides a probability $q_{n,y}$ that that function will be chosen when the algorithm needs a nonterminal with return type y. Similarly, for each terminal $t_y \in T_y$ the user provides

a probability $q_{t,y}$ that t_y will be chosen from T_y when the algorithm needs a terminal with return type y. The user also provides a maximum depth bound D, a maximum size bound S, a requested return type for the tree y_r, and a probability distribution of desired tree sizes $w_1, ..., w_S$ for each tree size from 1 to S.

Given:
maximum depth bound D
maximum size bound S
disjoint nonterminal subsets N_y of nonterminal set N for each $y \in Y$
disjoint terminal subsets T_y of terminal set T for each $y \in Y$
for each T_y and N_y,
probabilities $q_{n,y}$ and $q_{t,y}$ for each $t_y \in T_y$ and $n_y \in N_y$
return type for the tree $y_r \in Y$
probabilities $w_1, ..., w_S$ of generating a tree of size {1,...,S}
Do:
new tree **T** = StronglyTypedPTC2(y_r)

StronglyTypedPTC2(return type $y \in Y$)
Returns: a tree of depth $\leq D$, of return type y, and of size between 1 and $S + b_{max}$ inclusive, where b_{max} is the largest number of arguments to any nonterminal in N

Let r be a random tree size from the probability distribution $w_1, ..., w_S$
If $r = 1$ return a random terminal from T_y (by $q_{t,y}$ probabilities)
Else
Choose a nonterminal n_y from N_y (by $q_{n,y}$ probabilities)
Tree root $root \leftarrow n_y$
Node depth $d \leftarrow 1$
For each argument position a of n_y, Enqueue($\{a, d\}$)
Current tree size $s \leftarrow 1$
Until Size()$+s \geq r$ or Size()$= 0$,
{argument position a, depth d}$\leftarrow$RandomDequeue()
$y \leftarrow$argument type of a
If $d = D$, Fill a with a terminal t_y from T_y (by $q_{t,y}$ probabilities)
Else
Choose a nonterminal n_y from N_y (by $q_{n,y}$ probabilities)
Fill a with n_y
For each argument position a of n_y, Enqueue($\{a, d+1\}$)
$s \leftarrow s + 1$
Until Size()$= 0$,
{argument position a, depth d}$\leftarrow$RandomDequeue()
$y \leftarrow$argument type of a
Choose a terminal t_y from T_y (by $q_{t,y}$ probabilities)
Fill a with t_y
Return $root$

This algorithm works identically to PTC2, except that like StronglyTypedPTC1 it too must loosen the guarantees on probability of occurrence of nonterminals and terminals. Namely, each terminal t_y of a type $y \in Y$ will appear with $q_{t,y}$ probability relative to other terminals of type y, and each nonterminal n_y of type $y \in Y$ will appear with $q_{t,y}$ relative to other nonterminals of type y.

Like StronglyTypedPTC1, the complexity of this algorithm is dependent on the size of each set of functions by type, and the combinations of types of arguments to each function. However, the complexity is no worse than the complexity for PTC2. This is because the number of nonterminals that must be searched at any time is no more than $\|N\|$, and the number of terminals is no more than $\|T\|$. And like StronglyTypedPTC1, this algorithm can be adapted to STGP generic functions, with a complexity bounded by the maximum of the PTC2 complexity and $\|Y\|$.

XII. Subtree Mutation and Code Growth

One important issue is tree creation's role in subtree mutation ("point" mutation) as described in p. 105 [11]. Subtree mutation is an increasingly popular method for modifying GP trees to form new ones as part of its evolutionary process. In subtree mutation, a node is chosen randomly within a tree; the subtree rooted at this node is then replaced by a new, randomly-generated subtree. Commonly GROW is used to produce this new subtree.

As described below, it appears that even with no selection bias (trees selected for mutation are picked entirely at random from the population), repeated subtree mutation can naturally increase the average size of a GP population. There are provable examples showing trees growing naturally under subtree mutation using GROW or PTC1, even with no selection bias. Consider a function set of one nonterminal of arity 1, and one terminal, both equally likely to appear. The resultant "lists" have an E_{tree} of 2, but after applying subtree mutation to an initially-generated population of these "lists", the expected size of individuals in the population grows to 2.5 (shown in Theorem 6 below). Additionally, experimental results indicate that after repeated subtree mutation using either of these methods, the size of a tree consistently grows asymptotically towards some value. This has been tested for population sizes ranging from 1 to 100,000, with no selection bias.

It appears that (in the case of the "lists") if E_{tree} for newly mutated subtrees is restricted to 1.5, this will maintain $E_{tree} = 2$ for the population as a whole. A general solution for function sets other than the "list" function set described above is desirable; this could enable us to use PTC1 not only to guarantee an expected tree size for an initial population but maintain that tree size through subtree mutation. This would eliminate subtree mutation as one of several candidate culprits behind GP tree growth.

Lemma 2: If $-1 < x < 1$, then $\sum_{i=1}^{\infty} ix^i = \frac{x}{(1-x)^2}$.

Proof:

$$\sum_{i=1}^{\infty} ix^i = x + 2x^2 + 3x^3 + ...$$

$$x\sum_{i=1}^{\infty} ix^i = x^2 + 2x^3 + ...$$

Therefore,

$$(1-x)\sum_{i=1}^{\infty} ix^i = x + x^2 + x^3 + ... = \sum_{i=1}^{\infty} x^i$$

$$(1-x)\sum_{i=1}^{\infty} ix^i = \left(\sum_{i=0}^{\infty} x^i\right) - 1$$

From the Geometric Series, if $-1 < x < 1$, then

$$(1-x)\sum_{i=1}^{\infty} ix^i = \frac{1}{1-x} - 1 = \frac{x}{1-x}$$

$$\sum_{i=1}^{\infty} ix^i = \frac{x}{(1-x)^2}$$

■

Theorem 6: Let the set of functions F consist of a single terminal and a single nonterminal of arity 1 with $p = \frac{1}{2}$ (e.g., functions which build "lists"). The natural expected tree size of trees built from this set is $E_{tree} = 2$; however, after a subjecting

the individuals in the initial generation to subtree mutation, the expected tree size rises (to $\frac{5}{2}$).

Proof: Since there are only two functions (a terminal with 0 children and a nonterminal with 1 child) with equal likelihood of being generated, then the expected number of children of a newly generated node is $g = \frac{1}{2}$. From Theorem 1, this means that $E_{tree} = 2$. Since $p = \frac{1}{2}$, a "list" of node size 1 should clearly occur in $\frac{1}{2}$ of all cases, node size 2 should occur in $\frac{1}{4}$ of all cases, node size 3 has a probability of occurrence $\frac{1}{8}$, and so on. In general, the probability that a "list" of node size n will occur in a population is $\frac{1}{2^n}$.

The expected new size of an individual of size n undergoing subtree mutation is the original size minus the expected loss ($\frac{n+1}{2}$ for a list) plus the expected size of the new subtree. For a list this comes to $n - \frac{n+1}{2} + 2$. Hence the expected size of individuals in a population after subjecting each to subtree mutation is

$$\sum_{n=1}^{\infty} \frac{1}{2^n}(n - \frac{n+1}{2} + 2) = \left(\frac{1}{2}\sum_{n=1}^{\infty}\frac{n}{2^n}\right) + \left(\frac{3}{2}\sum_{n=1}^{\infty}\frac{1}{2^n}\right)$$

From Lemma 2, the first term reduces to 1. From the Geometric Series, the second term reduces to $\frac{3}{2}$. Therefore the new expected size of an individual is $\frac{5}{2}$. ■

XIII. Conclusions

PTC1 and PTC2 are an advance over GROW and add robustness to an important part of genetic programming. Unlike other tree-creation algorithms which provide uniformly distributed tree structures but have high computational complexity, PTC1 and PTC2 provide uniform distribution of functions and have very low computational complexity. Both of these algorithms compare well GROW, which runs in $O(E_{tree})$ time, but permits E_{tree} to be infinite in many common configurations.

PTC1 guarantees trees will be generated around an expected tree size, but does not provide control over variance in size. If the function set demands continuous-valued probabilities of appearance, PTC1 runs in $\leq O(\lg(\|F\|)E_{tree})$ time, where $\|F\|$ is the number of total functions, and E_{tree} is the (finite) expected tree size. With reasonable constraints, PTC1 can run in $O(E_{tree})$ time.

PTC2 takes a user-provided probability distribution by tree size, and approximates generating trees from this distribution. PTC2 runs in $O((r + b_{max}) \times \max(lg(||N||), lg(||T||)))$, or $O(r + b_{max})$ with reasonable constraints, where r is the average tree size in the probability distribution, $||N||$ is the number of nonterminals and $||T||$ the number of terminals in the function set, and b_{max} is the largest number of children of any nonterminal in the function set.

Acknowledgments

My thanks to Jim Hendler, Lee Spector, David Rager, Jim Reggia, Samir Khuller, James Owings, and Glen Henshaw for their suggestions and help in the preparation of this paper. Thanks also to the reviewers for their insightful comments and careful scrutiny.

References

[1] Angeline, P.J. Subtree Crossover Causes Bloat. *Genetic Programming 1998: Proceedings of the Third Annual Conference (GP98)*. J. Koza *et al*, editors. 745–752. San Fransisco, CA: Morgan Kaufmann, 1998.

[2] Blickle, T. and L. Thiele. Genetic Programming and Redundancy. *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94)*. J. Hopf, editor. 33–38. Max-Planck-Institut für Informatik, 1994.

[3] Böhm, W. and A. Geyer-Schulz. Exact Uniform Initialization for Genetic Programming. *Foundations of Genetic Algorithms 4*, R. Belew and M. Vore, editors. San Mateo: Morgan Kaufmann, 1997.

[4] Chellapilla, K. Evolving Programming with Tree Mutations: Evolving Computer Programs without Crossover. *Genetic Programming 1997: Proceedings of the Second Annual Conference (GP97)*. J. Koza *et al*, editors. 431–438. San Fransisco: Morgan Kaufmann, 1997.

[5] Chellapilla, K. A Preliminary Investigation into Evolving Modular Programs without Subtree Crossover. *Genetic Programming 1998: Proceedings of the Third Annual Conference (GP98)*. J. Koza *et al*, editors. 23–31. San Fransisco, CA: Morgan Kaufmann, 1998.

[6] Daida, J.M. *et al*. What Makes a Problem GP-Hard? Analysis of a Tunably Hard Problem. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO99)*. W. Banzhaf *et al*, editors. 982–989. San Fransisco: Morgan Kaufmann, 1999.

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

[8] Goldberg, D.E. and U.-M. O'Reilly. Where Does the Good Stuff Go, and Why? How Contextual Semantics Influences Program Structure in Simple Genetic Programming. *Proceedings of the First European Conference on Genetic Programming*. W. Banzhaf *et al*, editors. Berlin: Springer-Verlag, 1998.

[9] Haynes, T., D. Schoenfeld, and R. Wainwright. Type Inheritance in Strongly Typed Genetic Programming. *Advances in Genetic Programming 2*, P. Angeline and K. Kinnear editors. 359–375. Cambridge, MA: MIT Press, 1995.

[10] Iba, H. Random Tree Generation for Genetic Programming. *Parallel Problem Solving from Nature (PPSN) IV*. H.M. Voigt *et al*, editors. Springer, 1996.

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

[12] Langdon, W.B. and R. Poli. Fitness Causes Bloat. *Second On-line World Conference on Soft Computing in Engineering Design and Manufacturing*. P.K. Chawdhry, R. Roy, and R.K. Pan, editors. London: Springer-Verlag, 1997.

[13] Langdon, W.B. and R. Poli. Fitness Causes Bloat: Mutation. *Late Breaking Papers at the GP-97 Conference*. J. Koza, editor. 132–140. Stanford, CA: Stanford Bookstore, 1997.

[14] Luke, S. and L. Spector. A Comparison of Crossover and Mutation in Genetic Programming. *Genetic Programming 1997: Proceedings of the Second Annual Conference (GP97)*. J. Koza *et al*, editors. 240–248. San Fransisco: Morgan Kaufmann. 1997.

[15] Luke, S. and L. Spector. A Revised Comparison of Crossover and Mutation in Genetic Programming. *Genetic Programming 1998: Proceedings of the Third Annual Conference (GP98)*. J. Koza *et al*, editors. 208–213. San Fransisco, CA: Morgan Kaufmann, 1998.

[16] Montana, D.J. Strongly Typed Genetic Programming. *Evolutionary Computation*. 3(2):199–230. Cambridge, MA: The MIT Press, 1995.

[17] O'Reilly, U.-M. and F. Oppacher. Using Building Block Functions to Investigate a Building Block Hypothesis for Genetic Programming. Working Paper 94-02-029. Santa Fe Institute, 1994.

[18] O'Reilly, U.-M. The Impact of External Dependency in Genetic Programming Primitives. *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation*. 306–311. Piscataway: IEEE Press, 1998.

[19] Soule, T., J.A. Foster, and J. Dickinson. Code Growth in Genetic Programming. *Genetic Programming: Proceedings of the First Annual Conference, 1996 (GP96)*. J. Koza *et al*, editors. 215–223. Cambridge, MA: The MIT Press, 1996.

[20] Whigham, P.A. Search Bias, Language Bias, and Genetic Programming. *Genetic Programming 1996: Proceedings of the First Annual Conference (GP96)*. J. Koza *et al*, editors. 230–237. Cambridge, MA: The MIT Press, 1996.

[21] Zhang, B.-T. and H. Mühlenbein. Balancing Accuracy and Parsimony in Genetic Programming. *Evolutionary Computation*. 3(1):17–38, 1995.