An Arithmetic Test Suite for Genetic Programming

Dan Ashlock * Jim Lathrop †

April 2, 1996

Abstract

In this paper we explore a number of ideas for enhancing the techniques of genetic programming in the context of a very simple test environment that nevertheless possesses some degree of algorithmic subtlety. We term this genetic programming environment plus-one-recall-store (PORS). This genetic programming environment is quite simple having only a pair of terminals and a pair of operations. The terminals are the number one and recall from an external memory. The operations are a unary store operation and binary addition, +, on natural numbers. In this paper we present the PORS environment, present a mathematical description of its properties, and then focus on testing the use of Markov chains in generating, crossing over, and mutating evolving programs. We obtain a surprising indication of the correct situations in which to use Markov chains during evolutionary program induction.

*Mathematics Department Iowa State University, Ames, IA, 50010, email: danwell@iastate.edu

†Computer Science Department, Iowa State University, Ames Iowa, 50010, email: jil@iastate.edu. This research was supported in part by National Science Foundation Grant CCR-9157382, with matching funds from Rockwell International, Microwave Systems Corporation, and the Amoco Foundation.

1 Introduction

1.1 A Brief Introduction to Genetic Programming

In this paper we will introduce a test environment for genetic programming systems. A *genetic programming system* is software that is used to maintain a population of evolving computer programs, usually stored as parse trees. This population is generated at random initially, then improved by evolution until resources run out or an acceptable program appears in the population. In this context, *evolving* is meant in a sense similar to the biological one. Programs or parse trees that are more fit are allowed to "have children" that displace less fit programs. The famous evolutionary theory of Charles Darwin suggests that, over time, more nearly fit programs will appear. Genetic programming is, in essence, a biologically inspired program induction technique.

There are important differences between biological evolution and genetic programming. In a biological environment an individual's fitness is measured by the degree to which it manages to reproduce. In the artificial evolution of genetic programming the number of children an individual program has is determined from an abstract fitness heuristic which measures the program's performance. In a biological environment, the fitness ranking of an individual is based on its genetics, environment, and a healthy dollop of luck. In the artificial selection used in this paper, fitness is found with a function chosen to match the programming task that maps the parse trees in our population into the real numbers.

An example of a parse tree (labeled with the values computed at each step) is shown in figure 1. The parse tree represents the program that could be described in English as follows: "Add one and one, store the result in memory, then add what you stored to the result of recalling the contents of memory." When space is at a premium we may also use *LISP-like* notation in which the tree in figure 1 would read

(+ (Sto (+ 1 1)) Rcl).

The parse trees used in genetic programming are rooted trees in the combinatorial sense. The vertices of the trees are program statements with the leaves called *terminals* and the interior vertices called *operations*. Taken together the terminals and operations of a parse tree are called *nodes*. To execute a program, each operation, starting with the root, is executed recursively on the values returned by its sub-trees. Terminals return immediate

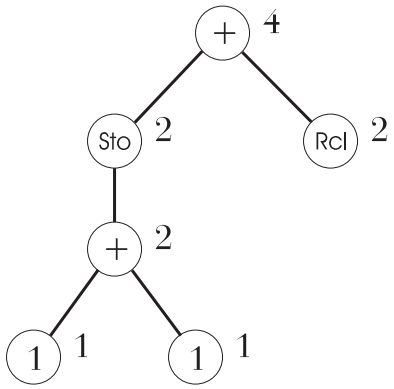


Figure 1: An example of a PORS parse tree

values, either constants or arguments of the program. The value returned by the root operation is the value returned by the entire program.

Within a genetic programming environment the evolving programs are typically in a special programming language made up for the problem under assault. It is impractical to use standard programming languages such as C, Pascal, C++, or Fortran for genetic programming because almost all randomly generated programs in these languages are not syntactically valid. A special purpose language can be designed so that there is a set of random programs, closed under whatever operations we use to drive our evolution, that are all syntactically valid. In addition to solving the problem of program syntactic validity, a specially designed language can be made to be more likely to contain solutions to the problem being treated. The designer simply includes appropriate operations. Our special purpose language is described in section 1.2.

A program in a genetic programming environment is not restricted to a single parse tree. The practice of using *automatically defined functions* (ADFs) gives genetic programs a structure equivalent to the subroutines and procedures of more standard programming environments. These subroutines or functions are stored as auxiliary parse trees and called by the original parse tree as operations. During evolution, an ADF operation is used like any other in the "main" parse tree. Whenever an operation associated with an ADF is called, the parse tree for the ADF is executed.

ADFs are only one possible subroutine-like structure for genetic programming. There is another such structure. In their paper on co-evolving high level representations Angeline and Pollack [1] define the process of *module acquisition* in which tree fragments which are used by many parse trees are transformed into new operations dynamically during evolution. The tree fragments are saved in a library and some effort is spent deciding when to remove modules from the library.

The details of the genetic programming system we will use are contained in the experimental descriptions later in the paper. We are using standard analogs to sex and mutation of the sort described in Koza's foundational text for genetic programming [3]. Readers interested in additional discussion and examples of genetic programming should consult [2].

1.2 The Test Environment

Although the PORS test environment has very few operations and terminals, it contains both easy and hard problems for use in testing the performance of genetic programming environments. The language has two terminals, the number 1 and a recall command. The recall command reports the contents of an external storage location, like the memory of an inexpensive pocket calculator. There are two operations, a store command that takes a single argument, the value of which is stored in the external memory and returned to the ancestor of the store operation, and the binary operation of addition with the usual definition. An example of a parse tree that computes the number 4 in this simple genetic programming language is shown in Figure 1. The tree in Figure 1 is labeled with the numbers returned by the various subtrees.

We do not use automatically defined functions but rather have a notion of *macros*. Macros amount to adopting a particular point of view about code fragments that appear quite often in the course of evolution. The two most common macros in the PORS environment are shown in Figure 2. They take whatever value is computed in the parse tree **T** and multiply it by two and three respectively. These macros are the same as the modules discussed in Angeline and Pollack's work on coevolving high level representations [1], but we do not dynamically acquire them during the process of evolution. We define macros only to allow us to more easily discuss the structure of our parse trees.

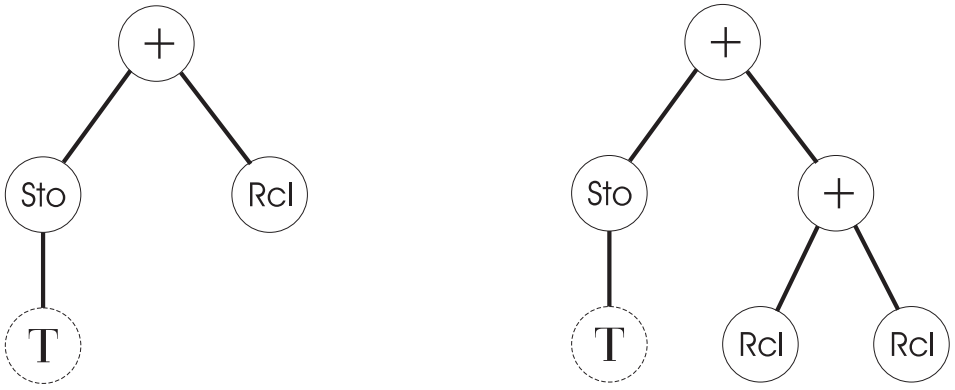


Figure 2: Subroutines for multiplying tree T by 2 or 3.

Since storing and recalling must occur in some order to have a well defined meaning, we adopt the standard that the left-hand branch of a parse tree is evaluated before the right-hand branch. In both the subroutines shown in Figure 2, this means that all the storing takes place before all the recalling (outside of the tree T). In a randomly generated or evolved parse tree there is no guarantee that a recall operation will not be requested before a store operation. To prevent this from being a problem, the external memory is initialized to 0 before a parse tree is evaluated.

2 The Test Problems

In the PORS environment, there are two very natural problems; one easy, one hard. The first problem, the easy one, which we term *efficient node use*, is to describe the largest possible number with a fixed number of nodes. The second problem, the difficult one, which we term *minimal description*, is to find the minimal number of nodes needed to describe a particular number. A bit of mathematical theory will help us find the true answer to the efficient node use problem and, with that in hand, we can solve some cases of the minimal description problem.

Let $\mathcal{T}$ be the set of all PORS trees. Let $\varepsilon : \mathcal{T} \to \mathbb{N}$ be the evaluation map that computes the number a tree describes. For a tree T, let $|T|$ denote

the number of nodes in the tree. Let $f(n)$ be the largest number that can be described by a PORS tree with n nodes. Call a tree T *optimal* if $\varepsilon(T) = f(|T|)$. For a tree T, denote by $\sigma(T)$ the contents of the storage register after the tree has been evaluated. Finally for a tree T denote by T' the result of replacing all the 1s in T with recalls.

Lemma 1 *Any optimal tree with six or more nodes contains a store instruction.*

Proof:

A tree without store instructions is a simple binary tree whose leaves are either recall or 1 and whose interior nodes are pluses. This forces an odd number of nodes so we need not consider trees on an even number of nodes. Without a store to put something other than zero in memory each recall contributes nothing. An optimal tree without any store instructions may, thus, be assumed to consist entirely of the terminal 1 and the operation plus. From this we see that such optimal trees contain $2k-1$ nodes and describe a value of k. Examine the trees in figure 3 with seven, eight, and nine nodes.

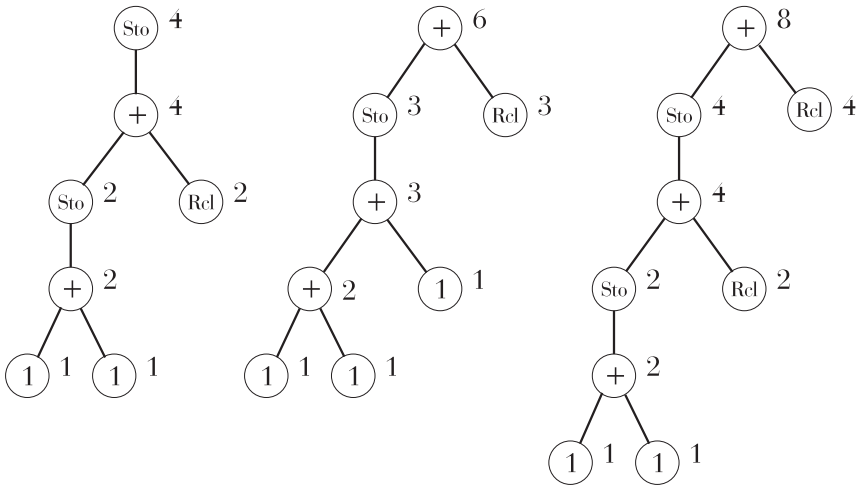


Figure 3: Good trees using the store instruction on 7, 8, and 9 nodes.

We see that these trees describe the numbers 4, 6, and 8 respectively and hence do at least as well as a store-free trees on seven, eight, and nine

nodes. By using the macro for multiplication by two, in Figure 2, we can extend these three examples to a family of trees that includes every odd number of nodes in excess of six and which describe numbers larger than the corresponding store-free trees. □

Lemma 2 *In an optimal tree, the right descendant of a plus may be assumed not to be a store.*

Proof:

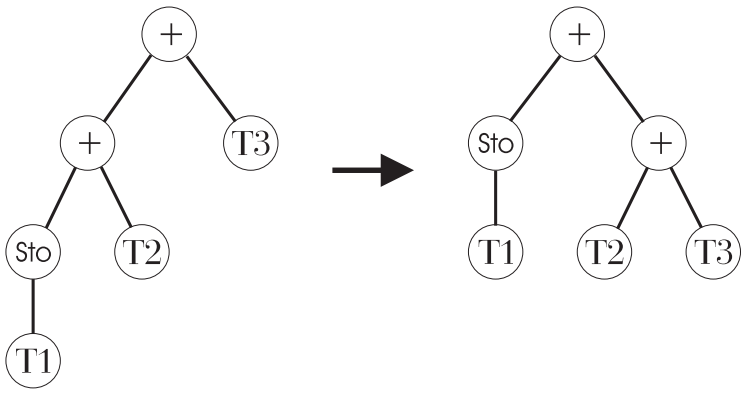
Suppose that the right descendant of a plus was a store. If we delete that store and insert a new store as the immediate ancestor of the plus in question, the value of the tree increases or stays the same without changing the number of nodes in the tree. □

Lemma 3 *An optimal tree that contains a store instruction may be assumed to contain a store instruction as the immediate left descendant of its topmost plus.*

Proof:

Let the subtrees branching off of the topmost plus in a tree be called the *major subtrees*. First, we claim that a tree containing a store contains a store in the left major subtree. To see this, assume we have an n-node optimal tree with a store but no store instructions in its left major subtree and let the tree have k nodes in its left major subtree and $n-k-1$ in its right major subtree. Clearly, the left major subtree is optimally composed of 1s and pluses. It hence returns $(k-1)/2$ for its value and has 0 in the memory when it finishes. Lemma 1 implies, thus, that the left major subtree contains at most five nodes and the fact such trees have an odd number of nodes forces the left major subtree to have one, three, or five nodes. In the right major subtree, we have a store instruction and, hence, a store instruction that is executed first. Since the tree is optimal, the argument of this store is itself an optimal store-free tree and, hence, has one, three, or five nodes. Now we have a pair of optimal, store-free trees, the left major subtree and the argument of the first store executed in the right major subtree. If we replace the left major subtree with *(Sto T)* where T is the larger of these two trees and put the smaller starting where the store had been, replacing all 1's in the right major subtree with recalls, then the value described by the tree

will not decrease. We may, thus, assume a store instruction is present in the left major subtree of an optimal tree. With the claim in hand we may now obtain the lemma by induction on the following transformation:



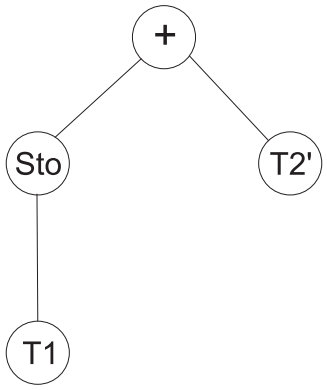
This transformation cannot decrease the value of a tree and clearly allows us to percolate store instructions up until one is the left descendant of the topmost plus in the tree. □

Theorem 1 *For $n \geq 6$ we have that*

$$f(n) = Max\left\{f(n-k-2)(f(k)+1) : 1 \leq k \leq (n-3)\right\}.$$

Proof:

Let T be an optimal tree on $n \geq 6$ nodes. By the hypothesis, T has at least six nodes and hence contains a store instruction by Lemma 1. We know from Lemma 3 that we may assume $T4$ has the form:



where $|T1| = (n-k-2)$, $|T2| = k$, and $T1$ and $T2$ are themselves optimal trees. A moments thought shows that because we replaced the 1s in $T2$ with recalls (this is the meaning of $T2'$) that

$$\varepsilon(T) = \varepsilon(T1) \cdot (\varepsilon(T2) + 1).$$

Since optimal trees on r nodes evaluate to $f(r)$ we have the theorem. □

This theorem, together with a small amount of easy hand enumeration, permits us to easily tabulate the values for $f(n)$, as in Figure 4. We do not yet have a nice closed form, but one is strongly implied by the entries of table.

n	f(n)	n	f(n)	n	f(n)
1	1	10	9	19	72
2	1	11	12	20	96
3	2	12	16	21	128
4	2	13	18	22	144
5	3	14	24	23	192
6	4	15	32	24	256
7	4	16	36	25	288
8	6	17	48	26	384
9	8	18	64	27	512

Figure 4: The first few values of $f(n)$.

3 Some Combinatorial Results

In order to explain the observed behavior in our genetic programming systems we need information about the space of PORS trees. To this end we derive some relevant counting formulae in this section.

Lemma 4 *The number of PORS trees on k nodes is*

$$\sum_{n=1}^{\left[\frac{k+1}{2}\right]} \frac{1}{n}\binom{2n-2}{n-1}2^n\binom{k-1}{2n-2}.$$

Proof:

Suppose we group the terminals *recall* and 1 together as leaves of our parse trees. If we ignore store instructions we obtain from a PORS tree an underlying binary tree with n leaves and $n-1$ internal nodes corresponding to pluses. Since we have k nodes in the PORS tree we have at least one leaf and at most $[\frac{k+1}{2}]$ leaves in this underlying binary tree. The index of summation in the formula above is over the possible number of leaves in the underlying binary tree. The Catalan numbers $C_n = \frac{1}{n}\binom{2n-2}{n-1}$ count the number of types of binary trees with n leaves giving the first term of the formula being summed. Once we know the leaf count and type of the underlying binary tree we must decide if each of the n leaves are *Rcl* or 1 yielding 2^n choices and the second term of the above summed formula. Finally we must place the store instructions in the tree. A store instruction may appear before the top plus in the tree or as the left or right descendant of any of the $n-1$ pluses in the tree yielding $2n-1$ total places a store could be placed in the tree. Any number of store instructions can be placed in each of these locations which makes the number of ways to place the store instructions a simple balls-in-bins problem. There are $k-2n+1$ nodes not in the underlying binary tree which must be stores and they must be placed without restriction in each of $2n-1$ locations which can happen

$$\binom{(k-2n+1)+(2n-1)-1}{(2n-1)-1} = \binom{k-1}{2n-2}$$

ways yielding the last term of the summed formula. □

Corollary 1 *The number of PORS trees on k nodes in which each leaf executed before the first store is executed is a 1 and each leaf executed after the first store is executed a recall is*

$$\sum_{n=1}^{[\frac{k+1}{2}]} \frac{1}{n}\binom{2n-2}{n-1}\binom{k-1}{2n-2}.$$

Proof: The counting formula is derived exactly as in Lemma 4 save that there is no choice in the identity of leaves. □

We will denote this restricted class of PORS trees by $\mathcal{T}^*$.

Lemma 5 *The number of PORS trees in which a store instruction never has a store instruction as an immediate descendant is*

$$\sum_{n=\lceil \frac{k+2}{4} \rceil}^{\lfloor \frac{k+1}{2} \rfloor} \frac{1}{n}\binom{2n-2}{n-1}2^n\binom{2n-1}{k-2n+1}.$$

Proof:

Adopt the notion of underlying binary tree from Lemma 4. Assume we are considering underlying binary trees with n leaves. The index of summation for the formula given in this lemma is still the number of leaves but now the $2n-1$ location in which a store may be placed must equal or exceed the number of stores. There are $k-2n+1$ stores so we see that $k-2n+1 \leq 2n-1$, and hence $n \geq \lceil \frac{k+2}{4} \rceil$, which verifies the index of summation. The next two terms are the Catalan numbers and number of ways to choose the leaves of the tree as in Lemma 4. When we place the store instructions, though, we have $2n-1$ locations available. Each location may receive at most one of the $k-2n+1$ store instructions. The choices involved are thus counted with a simple binomial coefficient $\binom{2n-1}{k-2n+1}$, and the formula is complete. □

We will denote this special class of PORS trees by $\mathcal{T}_s$.

Corollary 2 *The number of PORS trees in $\mathcal{T}^* \bigcap \mathcal{T}_s$ is*

$$\sum_{n=\lceil \frac{k+2}{4} \rceil}^{\lfloor \frac{k+1}{2} \rfloor} \frac{1}{n}\binom{2n-2}{n-1}\binom{2n-1}{k-2n+1}.$$

Proof: The counting formula is derived exactly as in Lemma 5 save that there is no choice in the identity of leaves. □

We will denote $\mathcal{T}^* \bigcap \mathcal{T}_s$ by $\mathcal{T}_s^*$.

Each of the three restricted families of parse trees described above has special properties that are enjoyed by optimal trees. To see this read carefully the proof of Lemma 3. If we start our search for an optimal trees within these families we make discovery of optimal trees more likely. Since the counting formulae derived above do not induce in a non-combinatorialist an immediate sense of how fast these functions grow we show the numerical values for enumeration of trees on $1 \leq k \leq 16$ nodes in Figure 5.

k	$\mathcal{T}$	$\mathcal{T}^*$	$\mathcal{T}_s$	$\mathcal{T}_s^*$	$\mathcal{T}_s^*/\mathcal{T}$
1	2	1	2	1	0.500
2	2	1	2	1	0.500
3	6	2	4	1	0.167
4	14	4	12	3	0.214
5	42	9	28	5	0.119
6	122	21	84	11	0.090
7	384	51	240	25	0.065
8	1206	127	720	55	0.046
9	3922	323	2208	129	0.033
10	12914	835	6848	303	0.023
11	43190	2188	21616	721	0.017
12	145950	5798	68880	1743	0.012
13	498170	15511	221744	4241	0.009
14	1714926	41835	719696	10415	0.006
15	5940014	113634	2352384	25761	0.004
16	20712646	310572	7737600	64095	0.003

Figure 5: Enumeration of various types of PORS trees.

4 Markov Chains for Efficient Node Use

In this section, we will report experimental results for solving the efficient node use problem with genetic programming. Since this problem is solved much more efficiently by Theorem 1 the solution to the problem isn't the point. Rather we wish to use different Markov processes to generate the initial population and check to see to what degree this enhances or impairs solution of the efficient node use problem by the genetic algorithm. Thus far in genetic algorithms people have tried to enhance their original population and mutation operators by choosing something other than a uniform distribution on the nodes that make up their random parse trees. An example of this appears in Teller's experiment with evolved controllers for virtual bulldozers [6]. The bias in the probability of selection of various program nodes can be viewed as containing some knowledge Teller had about the problem he was trying to solve.

Using Markov chains extends this idea. Using a nonuniform distribution to select the nodes of a parse tree is a zeroth order structure with no dependence on ancestry. A Markov chain allows higher order bias of selection of nodes. The selection of nodes is allowed to have dependence on history. The restricted classes of parse trees in Section 3 can all be generated by Markov processes. Details of the algorithmic implementation of the Markov processes follow.

4.1 Experiment Description

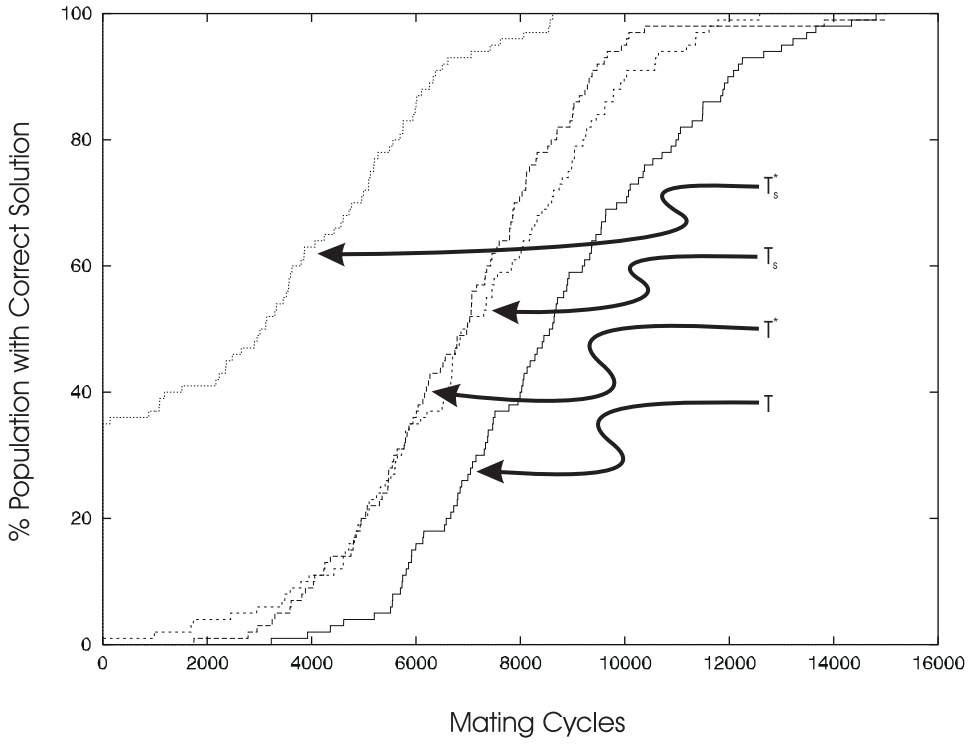
We will use four different methods of generating the initial population, corresponding to sampling from $\mathcal{T}$,$\mathcal{T}^*$,$\mathcal{T}_s$, and $\mathcal{T}_s^*$. The initial population will consist of 500 parse trees with exactly k nodes. We will run a steady state genetic algorithm until a parse tree that correctly solves the maximization problem on k nodes is found or until we have completed 25,000 mating events whichever comes first. A mating event consists of breeding two parse trees to produce two new parse trees, a total of four, and then placing the two most fit of the four in place of the two originally chosen. This is a strongly eleitist mating scheme.

Steady state genetic algorithms are described very well by Reynolds [4] and were discovered independently by Syswerda [5] and Whitley [7]. We decided to use the steady state algorithm because we are measuring success by computing evolutionary time until a correct answer appears in our population. A steady state algorithm gives much finer time resolution than a standard genetic algorithm with discrete, simultaneous generations.

A breeding event consists of four steps. First we copy the two original trees. Second we exchange uniformly chosen sub-trees of these copies (crossover). Next, with 50% probability for each of the new trees, we replace a uniformly chosen subtree with a sub-tree of the same size chosen uniformly from $\mathcal{T}$ (mutation). Mutation is done with new sub-trees taken from $\mathcal{T}$ in all four experiments - the Markov processes are used only to generate the initial population. Finally, if either of the new trees have in excess of k nodes we iteratively choose an immediate descendant of the root node to replace the tree until it has k or fewer nodes. We term this last process *chopping* and it is not yet standard in genetic programming.

For each of the four methods of generating the initial population we ran 100 initial populations and record the fraction that have found a correct

Figure 6: Fraction of populations with correct answer as a function of thousands of mating events with $K = 16$ nodes.



solution as a function of the number of mating events. The is results are shown in Figure 6.

The algorithms for generating our initial populations are as follows.

$\mathcal{T}$ We generate trees in $\mathcal{T}$ recursively according to the following rules. A tree on $k = 3$ or more nodes is uniformly chosen to be either a store with a $k - 1$ node tree as an argument or a $+$ with the remaining $k - 1$ nodes split between its left and right arguments by generating a uniformly distributed random number. A tree on $k = 2$ nodes is of the form store with a one node tree as an argument. Trees with $k = 1$ nodes are chosen to be 1 or recall with equal probability.

$\mathcal{T}^*$ We generate trees in $\mathcal{T}^*$ recursively according to the following rules. A tree on $k = 3$ or more nodes is uniformly chosen to be either a store with a $k-1$ node tree as an argument or a + with the remaining $k-1$ nodes split between its left and right arguments by generating a uniformly distributed random number. A tree on $k = 2$ nodes is of the form store with a one node tree as an argument. Trees with $k = 1$ nodes are chosen to be 1 if the leaf in the tree is not in a position where a store instruction has been generated otherwise it is chosen to be a recall.

$\mathcal{T}_s$ We generate trees in $\mathcal{T}_s$ recursively according to the following rules. A tree on $k = 4$ or more nodes that is not the immediate descendant of a store node is uniformly chosen to be either a store with a $k-1$ node tree as an argument or a + with the remaining $k-1$ nodes split between its left and right arguments by generating a uniformly distributed random number. If a tree on four or more nodes is the immediate descendant of a store node then it is a plus with the remaining nodes uniformly divided between its descendants. A tree on $k = 3$ nodes is a + with two one node trees as descendants. A tree on $k = 2$ nodes is of the form store with a one node tree as an argument. Trees with $k = 1$ nodes are chosen to be 1 or recall with equal probability.

$\mathcal{T}_s^*$ We generate trees in $\mathcal{T}_s^*$ recursively according to the following rules. A tree on $k = 4$ or more nodes that is not the immediate descendant of a store node is uniformly chosen to be either a store with a $k-1$ node tree as an argument or a + with the remaining $k-1$ nodes split between its left and right arguments by generating a uniformly distributed random number. If a tree on four or more nodes is the immediate descendant of a store node then it is a plus with the remaining nodes uniformly divided between its descendants. A tree on $k = 3$ nodes is a + with two one node trees as descendants. A tree on $k = 2$ nodes is of the form store with a one node tree as an argument. Trees with $k = 1$ nodes are chosen to be 1 if the leaf in the tree is not in a position where a store instruction has been generated otherwise it is chosen to be a recall.

5 Discussion of experimental results

As we can see see in figures 6 with $k = 16$ nodes the Markov chains were uniformly helpful. The worst performance is in the populations initially drawn from $\mathcal{T}$, the populations initially drawn from $\mathcal{T}^*$ and $\mathcal{T}_s$ have performance plots that repeatedly cross one another, and the populations initially drawn from $\mathcal{T}_s^*$ are substantially better than all three other sets of populations. There are a few oddities, for example the populations initially drawn from $\mathcal{T}_s$ included a few populations that had a hard time converging. As we will see soon this is likely because they had fallen into a large local optimum.

Another measure of the effect of the Markov chains is the number of populations that contained a solution in the initial population of which failed to find the solution in 25,000 mating events. In Figure 7 we tabulate both these for each of the four different types of initial populations.

k	$\mathcal{T}$		$\mathcal{T}^*$		$\mathcal{T}_s$		$\mathcal{T}_s^*$	
	Initial Success	Final Failure	Initial Success	Final Failure	Initial Success	Final Failure	Initial Success	Final Failure
6	74	0	100	0	100	0	100	0
7	100	0	100	0	100	0	100	0
8	43	0	100	0	99	0	100	0
9	4	0	46	0	21	0	100	0
10	3	0	78	0	43	0	100	0
11	1	0	40	0	20	0	100	0
12	0	5	1	2	1	6	12	7
13	0	0	15	0	7	0	99	0
14	0	0	2	0	2	0	38	0
15	0	29	0	32	0	50	0	58
16	0	0	1	0	0	0	35	0

Figure 7: Populations (out of 100) that contained a solution in the initial randomly generated population and which failed to find a solution.

If Figure 7 we see that the experiments with $k = 15$ nodes show the Markov generation of initial populations to result in degraded performance. The experimental probability of a population will not find a solution in 25,000

mating events goes from 0.26 to 0.58 as we add Markov generation to our genetic algorithm. A closer look at the table will show that $k = 15$ is an extreme example of another odd effect. While the efficient node use problem gets harder to solve with a genetic algorithm as the number of nodes increases it also shows a dependence of difficulty on the congruence class of the number of nodes (mod 3). There is a good explanation for this *a priori* bizarre feature of the problem.

Examine Figure 4. It's not too hard to see with a computer and Theorem 1 that the following suggestive facts hold if one has $k \geq 9$: $f(3n) = 2^n$, $f(3n+1) = 9 \cdot 2^{n-3}$, and $f(3n+2) = 3 \cdot 2^{n-1}$. A factor of three can come from the macro $3x$ or from a tree of the form (+ (+ 1 1) 1) while a factor of two can come from the macro $2x$ or a tree of the form (+ 1 1). In addition, a three of either sort has two possible forms: (+ (+ 1 1) 1) or (+ 1 (+ 1 1)). The form of either sort of two is unique. In all of our experimental runs, every efficient node use solution is made of twos and threes of the sort specified above. Keeping all this in mind this means there is a *unique* solution to the efficient node use problem on $3n$ nodes. When we have $3n + 1$ nodes the answer contains n "factors" two of which are threes and the rest of which are twos. There are $\binom{n}{2}$ ways to order the factors and two different forms the threes can have for a total of $2\dot{n}(n-1)$ solutions to the efficient node use problem. On $3n + 12$ nodes we have n factors with one three giving a total of $2n$ solutions. This variation in the size of the global optima of the search space in step with the congruence class *(mod 3)* goes a long way toward explaining the observations reported in Figure 7.

This is also bodes well for the efficient node use problem as a test problem for genetic programming environments. There are three families of problems within the efficient node use problem corresponding to the congruence classes (mod 3). These problems have markedly different fitness landscapes. Consider, for example, the local optima we alluded to previously when $k = 15$. The local optima contains trees that evaluate to the number 27. Producing a factor of three requires five nodes while produce a factor of two requires three nodes (see Figure 2). The macros that produce three have two variants while the macros that produce two have a unique form. This means the correct solution on fifteen nodes, a tree that evaluates to 32, is a unique of depth 10 while the trees that evaluate to 27 come in eight distinct forms and are of depth 9. Other multiples of fifteen nodes give search spaces with this same

pair of optima, powers of two and three, with the global optima remaining unique while the local optima grows exponentially in size with the number of nodes.

The three classes of problems within the efficient node use problem have a single point global optima ($k = 3n$ nodes), a global optima that grows quadratically ($k = 3n + 1$ nodes), and an optima that grows linearly ($k = 3n + 2$ nodes), all within an exponentially growing search space. This gives a fairly large set of well described test problems for use in evaluating a genetic programming environment.

6 A mathematical discussion of the minimal description problem.

We will let $m(k)$ be the minimum number of nodes in a PORS tree that evaluates to k. Out work on the efficient node use problem has already given us some information about the minimal description problem. Recall then $f(k)$ is is the largest number that can be described by a PORS tree on k nodes.

Lemma 6 *If $f(k) = n$ then $m(n) \leq k$.*

Proof:

This is obvious. □

Lemma 7 *If $f(k) = n$ and $s > n$ then $m(s) > k$.*

Proof:

If $m(s) \leq k$ then the minimal tree for producing s is a witness that we have a tree on k or fewer nodes that can produce a number bigger than n. This contradicts the hypothesis $f(k) = n$. □

Looking at base two expansions of integers can give a much meatier bound.

Lemma 8 *Suppose that the base two expansion of k contains ω ones and that $r = \lfloor \log_2(k) \rfloor$. Then*

$$m(k) \leq 3r + 2(\omega - 1).$$

Proof:

Recall the definition of the macro $2x$. Start with the parse tree

$$(2x\ (2x\ (2x \cdots\ (+\ 1\ 1) \cdots)))$$

for computing 2^r. For each one other than the most significant in the base two expansion of k break the parse tree between copies of $2x$ and insert a $+$ whose other argument is one. The result is a parse tree that computes k according to its Base 2 expansion. Each power of two requires three nodes, each inserted one requires two. □

Corollary 3 $m(k) \leq 5 \cdot \lfloor \log_2(k) \rfloor$.

Proof:

Adopting the notation of Lemma 8 note that $\omega - 1 \leq \lfloor \log_2(k) \rfloor$, substitute into the formula given in Lemma 8, and simplify. □

In Lemma 8 the base two expansion of n implies a nice construction using the macro $2x$ for a tree that computes n. The could be done in any base and in a base where a number has a sparse expansion (most digits zero) the upper bound may be better than the binary upper bound.

7 Markov Chains and Crossover

In this section we will explore a method for biasing crossover with a Markov chain. We define a modification of crossover operator that incorporates information from a Markov process and investigate the effect on the speed of convergence for various choices of the Markov process. The results are somewhat unexpected but can be explained in retrospect by appealing to the theoretical material developed in earlier sections. Simulations show that in some instances the Markov crossover operator increases the speed of convergence. In at least one instance, convergence is substantially slowed. In all cases the gain from enhancing the initial population exceed that of the Markov crossover operator but we conjecture this may be because of our choice of Markov process rather than any general property of evolutionary algorithms. We discuss possibly helpful modifications of the idea in the section on future work.

The Markov crossover operator requires that we weight the edges of each parse tree involved. Where before the Markov process we were using gave a probability distribution on successor nodes during the generation of trees in the initial population, we now use those probabilities to place weights on the edges of parse trees. We soften the distribution by displacing deterministic probabilities by a small amount, e.g. where the Markov process for generating a tree of a certain type had an edge that was disallowed, probability zero, we could place a weight of 0.05 on the type of edge in question. Likewise, edges that were required by the Markov process we might assign edge weights of 0.95 for use in the Markov Crossover. These weights are used as *binding strengths.*

With the binding strengths in hand, we perform Markov crossover as follows. In each of the two trees participating in the crossover we pick an edge, choosing with probability proportional to the reciprocal of the binding strengths. We then remove the subtrees starting below those edges and compute the binding strengths that would exist in the new trees formed, were we to complete crossover in the usual fashion. Independently for each subtree, we use these putative binding strengths to decide if we will attach the new subtrees. If a uniformly distributed random number is less than the computed binding strength then the new subtree is attached. If we do not attach the new subtree we generate a small random subtree using the Markov generation algorithm that inspired the binding strengths. In algorithmic form we would perform crossover of edge weighted parse trees T_1 and T_2 with binding strength function $BS(A, B)$ defined on pairs of nodes as follows:

1. For $i \varepsilon \{1, 2\}$ in T_i choose edges (A_i, B_i) with probability proportional to the reciprocal of $BS(A_i, B_i)$.

2. Compute $p_i = BS(A_i, B_{2-i})$ in the trees that would result in crossover with subtrees rooted at B_i.

3. With probability p_i complete the crossover in the standard fashion, independently for each value of i.

4. For each tree where crossover was not completed in the usual fashion, dispose of the unused subtree and generate a small new subtree with the same Markov generation technique that induced the binding strengths.

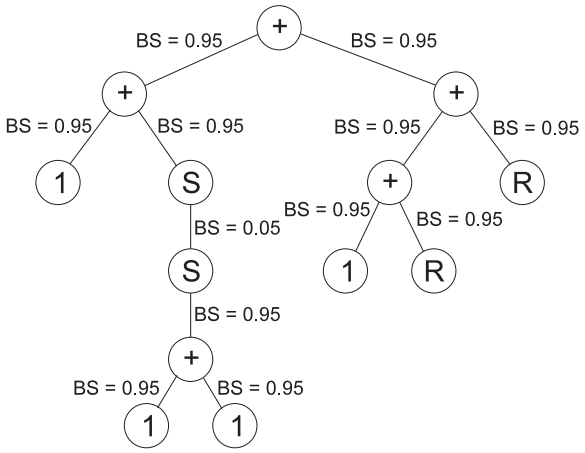


Figure 8: Tree with associated binding strengths

Intuitively the Markov crossover should have the same benefits of urging parse trees toward restricted classes that still contain correct solutions. The pressure toward restricted classes is uniform throughout evolution instead of being focused at the beginning of an evolutionary run, which may be good or bad depending on the cost/performance ratio. Figure 8 shows an example of a tree with its associated binding strengths. Figure 9 shows the associated probabilities and illustrates the selection of a subtree for crossover. An attempt is then made to attach the selected subtree to the crossover point on the other tree as shown in Figure 10.

In the remainder of the section we will report two experiments that test Markov crossover for particular choices of Markov process and hence of binding strength function. The first uses $\mathcal{T}_s$, the process in which the probability of a store following a store is zero and all other possibilities are equally likely whenever they are possible at all. For the $\mathcal{T}_s$ Markov process the binding strength function is:

$$BS_s(P,C) = \begin{cases} 0.05 & \text{if } P \text{ and } C \text{ are both STORE nodes} \\ 0.95 & \text{otherwise.} \end{cases}$$

The examples given in figures 8, 9 use this binding strength function. While not utterly forbidding a store to be the immediate descendant of a store this

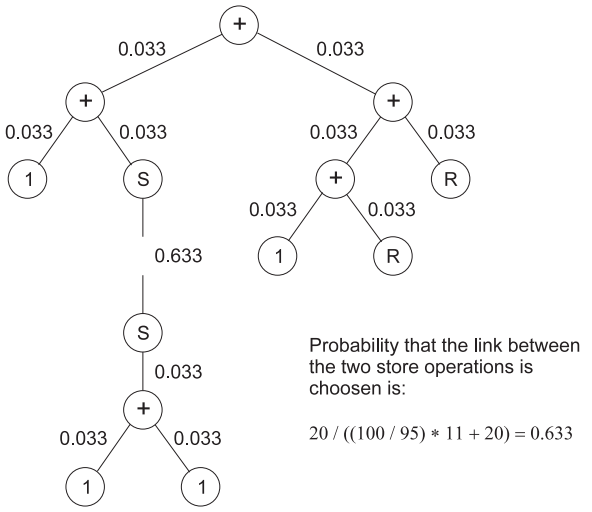


Figure 9: Tree with probabilities and selection shown

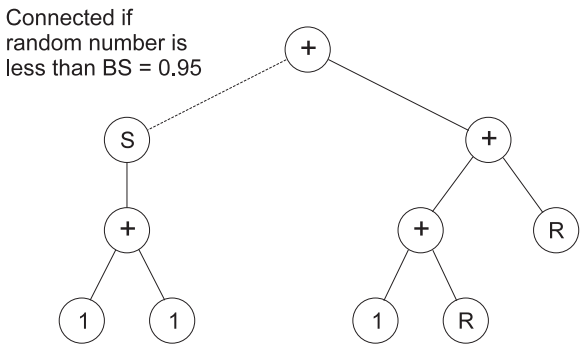


Figure 10: Binding a subtree with a tree

binding strength function greatly reduces the chance that two store nodes are executed one after another in a parse tree. In the earlier experiments reported in this paper crossover has no barrier beyond low fitness to joining a store with a store.

In Figure 11 we see the fraction of 500 populations, each consisting of 500 parse trees, that have found the correct solution as a function of mating events for four simulations. These simulations are a control in which the initial population and crossover are uniformly random, a Markov generated initial population with random crossover, a random initial population with Markov crossover, and a Markov initial population with Markov crossover.

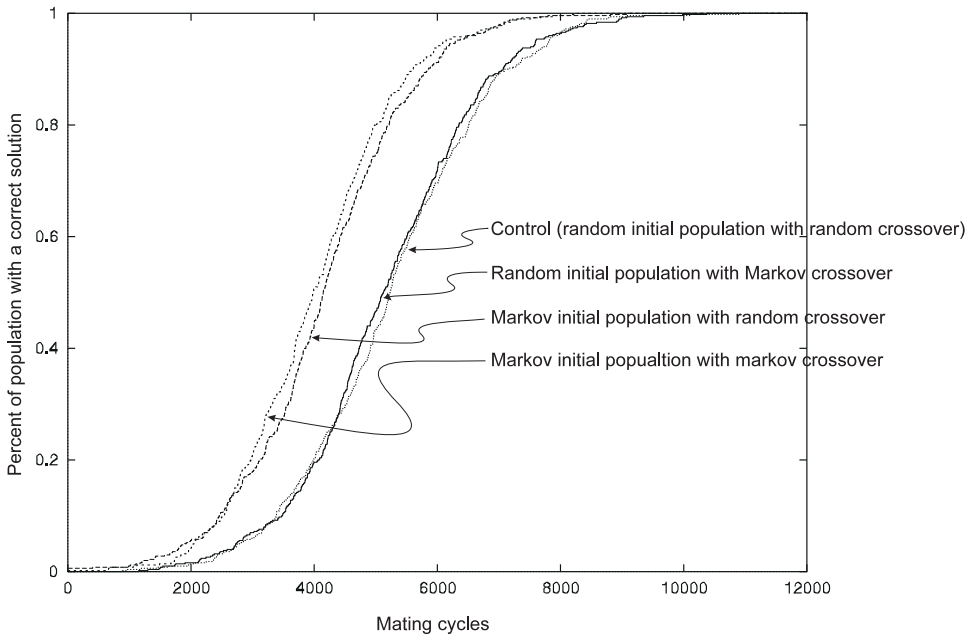


Figure 11: Graph of percent solutions versus generations of Markov T_s

Before performing this experiment we conjectured that Markov crossover would help a good deal more than Markov generation of the initial population but that placing the expensive computations involved in implementing the Markov crossover inside the innermost loop of the algorithm, in which breeding takes place, would be quite costly. We conjectured that we would

have to do a quantatative cost/benefit analysis and some hand wringing before issuing a judgment as to the worth of Markov crossover. Figure 11 speaks for itself. Most of the difference between runs in the experiment was due to Markov generation of the initial population. Since doing the Markov computations only during the generation of the initial population is overhead, swamped by the time spent doing simulated evolution, it is clear that Markov crossover is simply not worth the trouble. We make no conjecture that this is so outside of the PORS environment and have some thoughts, in section 8, as to better ways to do Markov crossover.

We now will do essentially the same experiment with the $\mathcal{T}_s^*$ Markov process, save that we will skip the trials using random generation with Markov crossover for reasons that will become apparent momentarily. For this Markov process we choose the binding strength function:

$$BS(P,C) = \begin{cases} 0.01 & \text{if adding the subtree causes the entire tree not to be in } \mathcal{T}_s \text{ and not in } \mathcal{T}^* \\ 0.05 & \text{if adding the subtree causes the entire tree to be in } \mathcal{T}_s \text{ but not in } \mathcal{T}^* \\ 0.05 & \text{if adding the subtree causes the entire tree to be in } \mathcal{T}^* \text{ but not in } \mathcal{T}_s \\ 0.95 & \text{if adding the subtree places the tree in } \mathcal{T}_s^* \end{cases}$$

In the last experiment there was little benefit from Markov crossover. To our surprise the Markov crossover in this experiment substantially impedes evolution. This can clearly be seen in Figure 12 where it is plotted against simulations where normal crossover is used. In retrospect there is a good explanation for this event.

Consider two parse trees that are in $\mathcal{T}_s^*$ and partition the nodes into two sets P and Q as follows. The set P of nodes are those executed before the first store instruction is executed and the set Q of nodes are those executed after the first store instruction is executed. In Figure 13 the nodes represented by circles belong to the set P and the nodes represented by squares belong to the set Q. Under $\mathcal{T}_s^*$ Markov crossover, a subtree from one tree will crossover normally, as opposed to causing a new subtree to be generated, with high probability only if all nodes in the subtree are from the same half of a $P - Q$ partition. It is not hard to see that for a 16 node tree such an event occurs with low probability; most subtrees contain nodes from both P and Q. Worse still, this probability of useful crossover decreases the tree

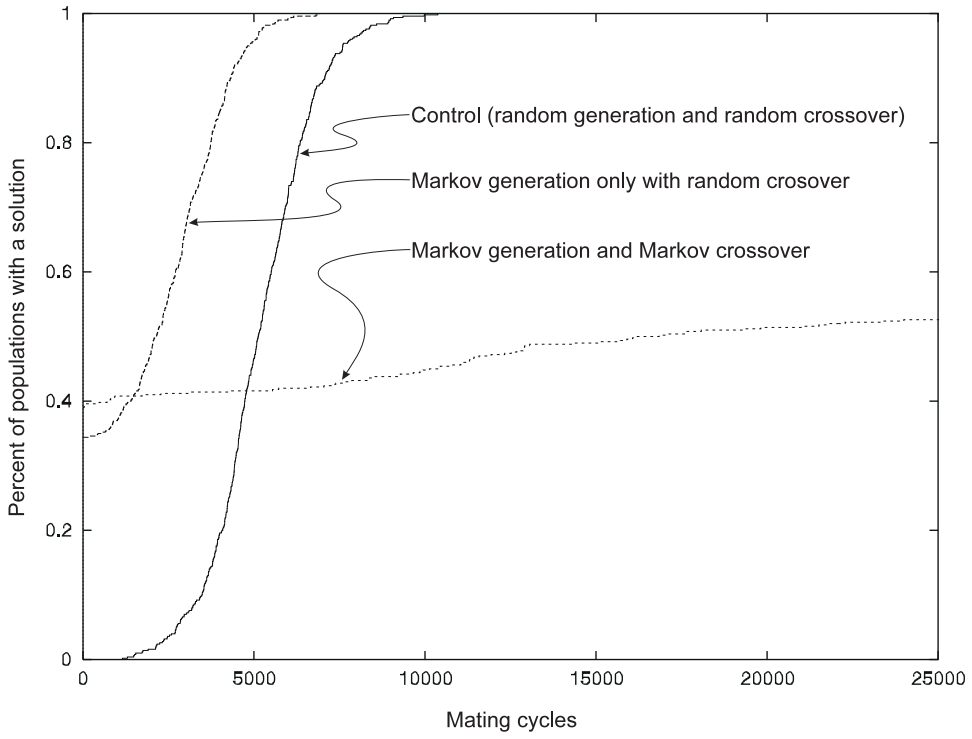


Figure 12: Graph of percent solutions versus generations of Markov T_s^*

approaches the left linear form of a correct solution. True crossover is thus rare in this implementation of Markov crossover while generation of a new, small subtree is common. Almost all useful work is performed by mutation, yielding very slow convergence time.

8 Future Work

The next step we wish to take in this research is to attempt to save an idea of which we are proud but which did not survive experimental testing in this paper; Markov crossover. It is possible to argue at great length that this or that Markov process might be *the* golden example that will provide a proof-of-concept for Markov crossover. Having tested several Markov crossover

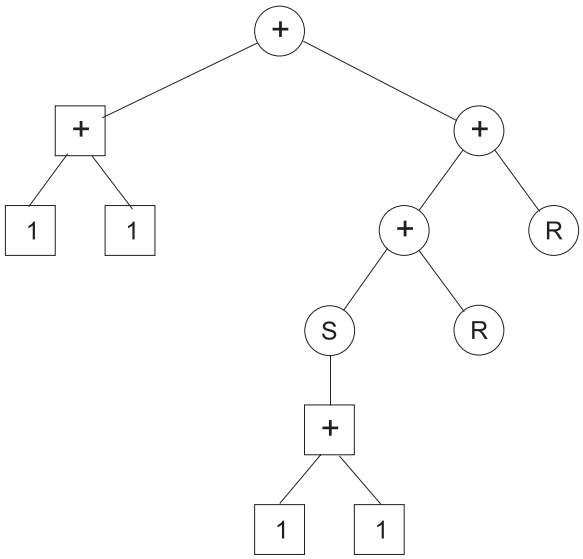


Figure 13: Partitioning a parse tree

operators, two of which were presented in this paper, we propose, instead of endless hacking, to put the problem back in the lap of Darwin. We draw out inspiration from molecular biology where the binding strength of various chemical bonds helps to dictate the pattern of activity of biological reactions. Those systems whose patters of strong and weak bonds are more efficient survive and reproduce. There is, in such a system, no need to design the patters on strength and weakness in the binding. These patterns are simply a gift given by evolution.

Generalizing the notion of Markov crossover as presented in this paper, we intend to build random crossover strengths into our parse trees. As a population improves in fitness, and declines in diversity, the binding strengths should converge to a small set of values which will suggest, in a natural fashion, a good Markov process. At least we conjecture this will be the case. With such a Markov process in hand we can both test the Markov process for use in generating new initial populations and we can check if the process scales to larger instances of the same problem. This latter idea deserves a bit more comment.

Suppose we are attempting to treat a given program induction problem with a genetic programming system. In addition, imagine that the problem comes in many sizes. It is not implausible that there are simple operations, built of but not contained in the primitive operations of the genetic programming system, that are useful in all or most of the instances of the target problem. Using parse trees with evolving binding strengths, we can discover these intermediate objects. They would appear as tree fragments with high internal binding strengths and lower binding strengths on their periphery. The location of such tree fragments was in fact the goal of Angeline and Pollacks technique of module acquisition. It is, to a lesser degree, the motive for including ADFs in a genetic programming system. Why, then, do we suppose this binding strength technique to be worth trying?

With module acquisition, the modules were, perhaps unfortunately, removed from the evolving portion of the code. In the environment we propose, the tree fragments would remain in the digital soup. The process of exploration would continue to operate on the fragments as well as the trees containing them. The use of binding strengths would transfer to evolution the job of deciding which code fragments are important enough to save, worth giving up, or in need of addition testing. The additional bookkeeping is reduced to a modified crossover operator, a substantial reduction in the support complexity.

References

[1] Peter J. Angeline and Jordan B. Pollack. Coevolving high-level representations. In Christopher Langton, editor, *Artificial Life III*, volume 17 of *Santa Fe Institute Studies in the Sciences of Complexity*, pages 55–71, Reading, 1994. Addison-Wesley.

[2] Kenneth Kinnear. *Advances in Genetic Programming.* The MIT Press, Cambridge, MA, 1994.

[3] John R. Koza. *Genetic Programming.* The MIT Press, Cambridge, MA, 1992.

[4] Craig Reynolds. An evolved, vision-based behavioral model of coordinated group motion. In Jean-Arcady Meyer, Herbert L. Roiblat, and

Stewart Wilson, editors, *From Animals to Animats 2*, pages 384–392. MIT Press, 1992.

[5] Gilbert Syswerda. A study of reproduction in generational and steady state genetic algorithms. In *Foundations of Genetic Algorithms*, pages 94–101. Morgan Kaufmann, 1991.

[6] Astro Teller. The evolution of mental models. In Kenneth Kinnear, editor, *Advances in Genetic Programming*, chapter 9. The MIT Press, 1994.

[7] Darrel Whitley. The genitor algorithm and selection pressure: why rank based allocation of reproductive trials is best. In *Proceedings of the 3rd ICGA*, pages 116–121. Morgan Kaufmann, 1989.