Chapter 8

Plus One Recall Store

In this chapter, we will do our first computer experiments with genetic programming. The first section is intended as a general introduction to genetic programming and is less narrowly focused than the rest of the chapter. It also contains no experiments, just preparatory reading and problems intended to build needed cerebral muscles. The second section introduces the Plus-One-Recall-Store (PORS) problems, previewed in Problems 1.14 and 1.15. The third section studies in detail the technique of population seeding, using nonrandom initial populations to enhance performance of an evolutionary algorithm. The fourth section applies various Alife techniques from earlier chapters to the PORS problems.

As an application, PORS is not terribly useful. One of the two main problems is solvable by theorem and proof techniques, so that the true optimum solutions are already known. (Technical details have been kept to an absolute minimum.) We study PORS not for itself but to learn genetic programming, much as we studied the string evolver in Chapter 2 to learn about evolutionary algorithms.

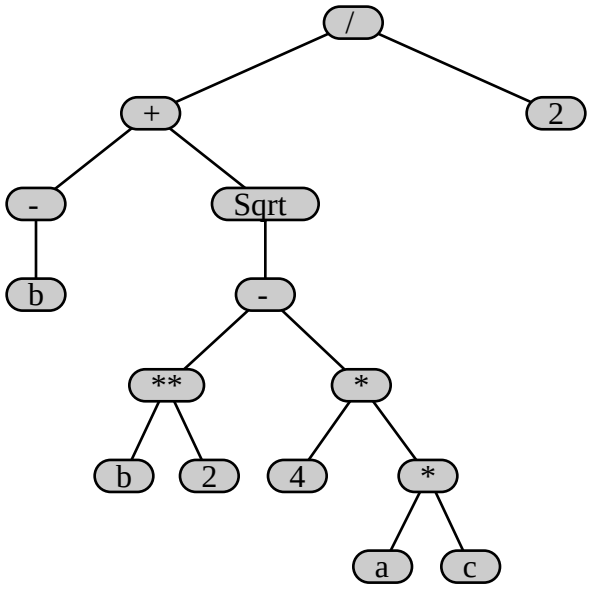
Leaving aside the practical applicability of the PORS problems, these problems have a deep and rich mathematical structure, reflected in the problems given in the chapter. Instructors of a mathematical bent may wish to extend these problems, particularly if they are in a combinatorial way. Engineering classes using this text should assign problems from this chapter with great caution; some require a good deal of familiarity with the various forms of abstract nonsense that so delight mathematicians.

8.1 Overview of Genetic Programming

Genetic programming (abbreviated GP) is defined to be the use of an evolutionary algorithm to evolve computer programs, typically stored as parse trees. A parse tree is shown in Figure 8.1 in tree form as well as in a functional notation used in the computer language *LISP*. We

term the compressed notation *Lisp-like*.

A parse tree has interior nodes, called *operations*, and leaves or terminal nodes, called *terminals*. Operations have the usual definition. In the tree shown in Figure 8.1, the operations are *Sqrt*, /, +, -, * and **. Terminals can contain external values passed to the program, input/output devices, or constants. The terminals in Figure 8.1 are the external inputs a, b, and c and the constants 2 and 4. The *root* of the tree is the first operation when the tree is written in Lisp-like form. In Figure 8.1, the root is the divide operation. The subtrees that are linked to an operation are called the *arguments* of the operation. The *size* of a parse tree is the number of nodes in it. The *depth* of a parse tree is the largest distance, in tree links, from the root to any terminal. The depth of the parse tree in Figure 8.1 is 6.



(/ (+ (- b) (Sqrt (- (b 2) (* 4 (* a c))))) 2)**

Figure 8.1: An example of a parse tree in standard and Lisp-like notation

Genetic Operations on Parse Trees

In genetic programming, crossover is performed by exchanging subtrees as shown in Figure 8.2, with parents above and children below. This crossover operation is called *subtree mutation*. Mutation consists of taking a randomly chosen subtree, deleting it, and replacing it with a new randomly generated subtree. In later chapters, we will add other mutation operators. This mutation operator is called *subtree mutation*. Notice that all notion of a "point mutation" as developed in preceding chapters has gone completely by the board. Mutation

at some loci of a parse tree has an excellent chance of changing large numbers of other loci (see Problem 8.11).

We will also use a third type of variation operator called *chopping* which reduces the size of a parse tree. The genomes we have used in previous chapters have had uniform size. A gene did not have a risk of become unmanageably large after several generations. The crossover operation used in genetic programming conserves the total number of nodes in the parents, but one child may get more and another fewer. As a result, the number of nodes in one creature can, theoretically, grow until it swamps the machine being used. The chop operator is used to reduce the size of a parse tree if mutation and crossover have made it "too large." In order to apply the chop operator, you select one argument of the root operation of a program and allow it to become the entire program, deleting the rest of the parse tree.

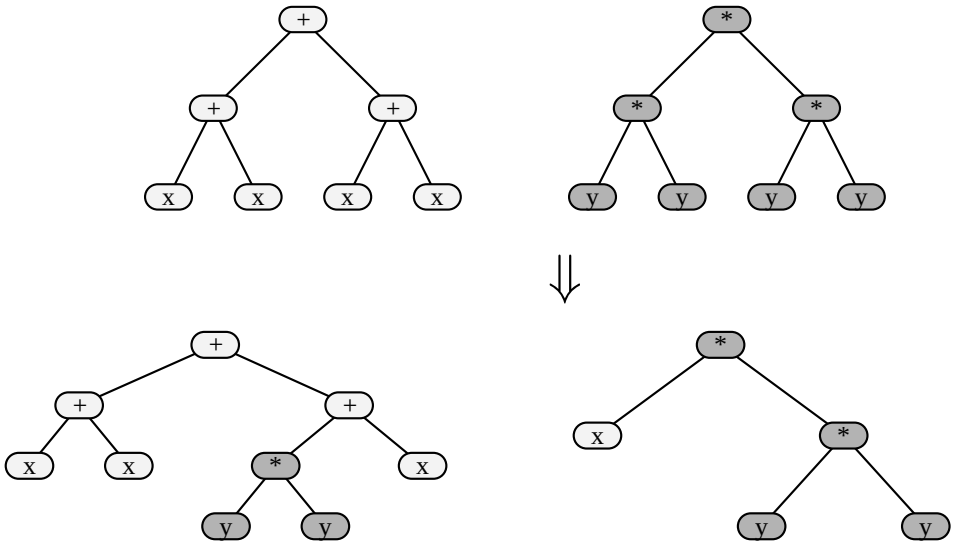


Figure 8.2: Crossover of two trees by exchanging subtrees

Genetic programming requires mature use of dynamic memory allocation. A parse tree is not declared as a single data structure but is rather built up recursively. The data structure used to store a node of a parse tree contains the node's type (constant, external input, unary operation, binary operation, etc.), the node's value (e.g., the numerical value of a constant or the identity of an external input), and pointers to the subtrees that form its arguments if it is an operation. Since the node data structure must be able to hold several different types of nodes, parts of its structure are often unused. If the node is a nonconstant terminal, the value field may be empty. If the node is a terminal, it will have all of its pointers empty. An example of a data structure for one node of a parse tree is shown in Figure 8.3.

```
struct node {float val;     //holds value for contant terminals
             type int;      //holds type - e.g. operation code
             node* args[2]; //pointers to argument trees
            };
```

C or C++

```
Node=^NodeRec;  {pointer to a node record}
NodeRec=Record
           val : Real;                   {value field}
           typ : Integer;                {operation/terminal type}
           args : Array[0..1] of Node;   {argument pointers}
        end; {NodeRec}
```

Pascal

Figure 8.3: Examples of data structures for holding nodes used to build parse trees

Generating Random Parse Trees

The first computational step in an evolutionary algorithm is to generate a random population, so we must learn to generate random parse trees. A random parse tree on n nodes with unary or binary operations is generated recursively as follows. If $n = 1$, then we generate a terminal node. If $n = 2$, we generate a unary operation, from those available, whose argument is a tree of size one. If $n \geq 3$, then we pick an operation from among those available. If that operation is unary, we generate its argument as a tree of size $n - 1$. If that operation is binary, we split $n - 1$ randomly into two nonzero pieces a and b and generate random trees of size a and of size b to be the arguments of the binary operation.

The random tree generation algorithm only knows about 3 sizes of trees, size one (terminal), size two (operation terminal), and size three or more (operation arguments).

Type Checking

Can you do genetic programming with real source code? Many of you may feel a chill running down your spine at the thought of what crossover and mutation might do to real source code. There are many structural reasons why genetic programming evolves limited, small parse trees instead of full computer programs. Most of these reasons are tied to the complexity and fragility of "real" computer code. For the most part we avoid this sort of problem by simply evolving small pieces of code that do very specific tasks. There is one source of fragility that we will avoid by main force: typing. Our genetic programs will have

a single type.

One of the most common errors a beginning programming student will make is a type check error. The student may read in a number as a character string and then be hurt and surprised when adding one to it causes an error (in any reasonable language) or when adding one mysteriously causes the first digit to vanish (in C or C++). To avoid having our evolving programs make (or spend all sorts of computer power avoiding) type errors, we will have only one type. This means that all terminals must return values of that type, and all operations must take as arguments values of that type and return values of that type.

Requiring that the parse tree have a single type in all its nodes does not completely deprive us of the advantages of having multiple types. Both C and BASIC have numerical Booleans. Zero is considered false and nonzero is considered true. This means that operations like $\leq$, $=$, or $\geq$ can be defined as numerical operations that, in some situations, function as boolean operations, all within a single data type. In this chapter, our data type will be integers.

Problems

Problem 8.1 *Transform each of the following expressions from functional form into Lisp-like notation. Do* not *simplify or modify the expressions algebraically before transforming them.*

(i) $f(x) = \frac{x^3}{x^3-1}$, (ii) $g(x) = (x-1)(x-2)(x-3)(x-4)$,

(iii) $h(x,y) = \sqrt{(x-a)^2 + (y-b)^2}$, (iv) $d(a,b,c) = \sqrt{b^2 - 4ac}$,

(v) $r(\theta) = 2 \cdot \cos^2(\theta)$, (vi) $r(\rho,\theta) = \sqrt[3]{\rho^2\theta^2 + 1}$.

Problem 8.2 *Transform each of the following expressions from Lisp-like notation into functional form. Do* not *simplify or modify the expressions algebraically before transforming them. Assume that logical connectives like "$<$" or "$\geq$" return 1 for true and 0 for false. You will need to name the functions. Assume x, y and z are variables and that any other letters or names represent constants. Notationally simplify the functions as much as you can.*

(i) **(+ (Sqr (Tan x)) 1)**

(ii) **(+ (+ (Sqr x) (Sqr y)) 1)**

(iii) **(+ (* (b $\geq$ 0) (/ x 2)) (* (b $<$ 0) (/ (- 0 x) 2)))**

(iv) **(/ (+ (+ x y) z) (+ (* x (* y z)) 1))**

(v) **(** (+ x (+ y z)) 3)**

(vi) **(/ 1 (+ 1 (* x x)))**

Problem 8.3 *On page 203, the claim is made that the crossover operation for parse trees, if repeated, can cause the size of the tree to grow without bound. Assuming we start with a population made of trees of the form* **(+ a b)** *in generation zero, compute the worst case size of the biggest tree in generation n. Give examples of worst case trees for* $n = 1, 2, 3,$ *and* 4*.*

Problem 8.4 *On page 204, there is a description of the algorithm for generating parse trees with operations that are unary and binary. Describe the algorithm for generating parse trees with unary, binary, and trinary operations. Also, show the data structure needed, as in Figure 8.3.(Use your favorite computer language.)*

Problem 8.5 *In order to generate parse trees, you must break a remaining number* $n - 1$ *of nodes* $(n \geq 3)$ *into 2 nonzero pieces a and b. Give pseudo-code for doing this.*

Problem 8.6 *In order to generate parse trees with trinary operations (see Problem 8.4), you must break a remaining number* $n - 1$ *of nodes into 3 nonzero pieces a, b, and c. Give pseudo-code for doing this.*

Problem 8.7 *Suppose we have a language for parse trees in which there are only terminals and binary operations. Prove all parse trees in this language have odd size.*

Problem 8.8 *The* Catalan numbers*,* C_n*, count the number of ways to group terms when you have a nonassociative operation on n variables. The first few values are* $C_1 = 1$*,* $C_2 = 1$*,* $C_3 = 2$*,* $C_4 = 5$*. The corresponding ways of grouping terms are:*

$n = 1$ (a),

$n = 2$ $(a \odot b)$,

$n = 3$ $(a \odot (b \odot c))$, $((a \odot b) \odot c)$,

$n = 4$ $(a\odot(b\odot(c\odot d)))$, $(a\odot((b\odot c)\odot d))$, $((a\odot b)\odot(c\odot d))$, $((a\odot(b\odot c))\odot d)$, $(((a\odot b)\odot c)\odot d)$.

First, find a general formula for the Catalan numbers. Second, explain why C_n *is also the number of parse trees with* $2n + 1$ *nodes, if the language for those parse trees has one binary operation and one terminal.*

Problem 8.9 *Suppose we have a GP-language with x terminals and y binary operations. Give a formula for the number of possible trees in terms of the number of nodes n in the tree. Modification of the answer to Problem 8.8 may serve.*

Problem 8.10 *Suppose we have a GP-language with x terminals, y binary operations, and z unary operations. Give a formula for the number of possible trees in terms of the number of nodes n in the tree. If you find this hard, you should first do Problem 8.9 or other cases where one of x, y, or z is 0. Your answer should involve a summation over something and may use the Catalan numbers.*

Problem 8.11 *The* rate *of a mutation operator for parse trees is the fraction of the nodes it will replace on average. Mathematically or experimentally, estimate the rate of a mutation operator that selects, with uniform probability, a node in a parse tree and replaces the subtree rooted there with a new subtree exactly the same size. If the GP-language in question has only unary operations, then this isn't too hard to compute. A language with binary operations is harder. A language with both unary and binary operations may well be too hard to do other than experimentally. Detail exactly your mathematical or experimental techniques.*

Problem 8.12 *Suppose we have a GP-language whose data type is integers. Give a definition of a binary operation that can be used as an If-then and of a trinary operation that can be used as an If-then-else. Discuss briefly the alternatives you considered and why you made the choices you did.*

Problem 8.13 *Suppose you were evolving real-valued parse trees that were encoding functions of one variable (stored in a terminal "x") to minimize the sum of squared error at 80 sample points with the fake bell curve (see Equation 2.1). Assume that your parse tree has available the operations of an inexpensive scientific calculator. Show mathematically why*

$$\textbf{(Sqr (Cos (Atan x)))} \textbf{ or } f(x) = Cos^2(Tan^{-1}(x))$$

is not an unreasonable result.

Problem 8.14 Essay. *Reread Problem 8.13. Clearly the parse tree:*

(/ 1 (+ 1 (* x x)))

would give perfect (zero) sum of squared error with points sampled from the fake bell curve. Why then did this parse tree never come up in 80 runs (done by the author) while the parse tree given in Problem 8.13 appeared in 5 of the runs? Consider the action of mutation and crossover of the parse trees when answering this question.

8.2 The PORS Language

The Plus-One-Recall-Store (PORS) language has two terminals, one unary operation, and one binary operation. They are summarized in Figure 8.4. The environment in which PORS

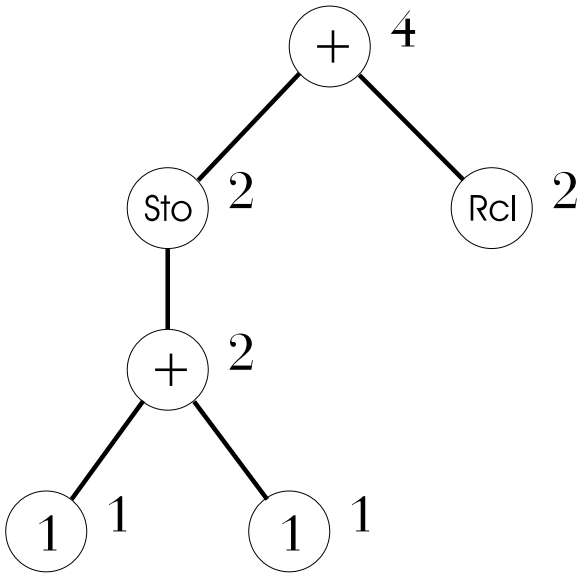
programs exist contains an external memory to which the program has access. Much like the memory in a cheap pocket calculator, the user may store and recall values to and from the memory. The binary operation of the PORS language is normal integer addition. The unary operation, **Sto**, does nothing to its argument, it just returns whatever value it is given. It is used for its side effect of storing the number that passes through it into the external memory. The terminal, **Rcl**, returns the value in the external memory. The terminal, **1**, is a simple integer constant.

Type	Name	Description
Terminal	1	integer constant, one
	Rcl	recalls an external memory
Unary operation	Sto	stores a value in an external memory and returns the number stored as its value
Binary operation	+	integer addition

Figure 8.4: Nodes in the Plus-One-Recall-Store language

Semantics of PORS

Now that we have specified the elements of a PORS parse tree, we can learn how to "run" it as a program. This is called *evaluating* the parse tree. The tree is evaluated from the bottom up. Since we are working with randomly generated parse trees, we will encounter trees that execute a recall instruction before their first store instruction. To prevent this from being undefined and hence a problem, declare the external memory location to be initialized to 0 at the beginning of the evaluation of any parse tree. The execution of the store and recall instructions and the constant 1 are obvious and unambiguous. When executing a plus, execute all the instructions in the left argument before those in the right argument. An example of a PORS parse tree, adorned with the values each node returns when executed, is shown in Example 8.1.



Example 8.1 *The parse tree shown above performs the following steps when evaluated: it adds 1 and 1, stores the resulting 2 in memory (also returning the 2 as the value of the* **Sto** *operation), and then adds that 2 to the contents of memory to obtain the result of the entire program, 4.*

For a PORS parse tree T, we denote by $\epsilon(T)$ the result of evaluating the parse tree (equivalently, running the program described by the tree). $\epsilon(T)$ is said to be the *value* of T. To start evolving parse trees, we lack only a problem to solve. We will work on the two tasks implicit in Problems 1.14 and 1.15. The *Efficient Node Use Problem* asks "what is the largest number that can be generated by a parse tree with n nodes?" The *Minimal Description Problem* asks, "given k, what is the smallest tree T for which $\epsilon(T) = k$?" The first problem is easy while the second is quite hard, as we shall see. Before attacking either of these problems, we need to get the basic routines for genetic programming built and working.

Experiment 8.1 *Write or obtain a set of routines for manipulating parse trees in the PORS language including all the following, as well as any others you find convenient.*

Name	*Argument*	*Returns*	*Description*
RandTree	*integer*	*pointer*	*generates a random tree with int nodes*
SizeTree	*pointer*	*integer*	*finds the number of nodes in a tree*
DeepTree	*pointer*	*integer*	*finds the depth of a tree*
CopyTree	*pointer*	*pointer*	*makes a copy of a tree, doing all needed dynamic allocation*
KillTree	*pointer*	*nothing*	*disposes of a parse tree*
PrinTree	*pointer*	*nothing*	*prints out a parse tree*
SaveTree	*pointer*	*nothing*	*saves a tree in a file*
ReadTree	*nothing*	*pointer*	*reads a tree from a file*
RSubTree	*pointer*	*pointer*	*returns a pointer to a random subtree of a tree*
CrosTree	*pointers*	*nothing*	*performs the "exchange subtrees" crossover operation used in genetic programming*
MuteTree	*pointer*	*nothing*	*picks a random subtree, deletes it, and replaces it with a random one the same size*
ChopTree	*pointer,integer*	*pointer*	*replaces a tree with one of its subtrees coming off the root, repeatedly, until the tree has int or fewer nodes*
EvalTree	*pointer*	*integer*	*computes the value of a tree*

When the above routines are ready, test them by computing the following for a population of 1000 trees of size 5, 10, and 20 respectively.

(i) mean and standard deviation of tree depth

(ii) a histogram of how often each depth occurred

(iii) a histogram of the values of the trees

Also, print out examples of copying, mutation, crossover, and chopping.

Experiment 8.1 is a baseline for later experiments in this chapter. For the most part, it is included so that you can test and debug the underlying routines. The software in this chapter uses dynamic allocation. Hence, it is more likely to contain subtle bugs than the software you coded in the preceding chapters. In our next experiment, we will put these routines to work attempting to solve the Efficient Node Use Problem for small n. The Efficient Node Use Problem has a natural fitness function: the evaluation function ϵ. So long as we keep the number of nodes in trees in our population bounded by the number of nodes, n, for which we are currently trying to solve the Efficient Node Use Problem, we can set the tree's fitness to its value.

n	$max\ \epsilon(T)$	n	$max\ \epsilon(T)$	n	$max\ \epsilon(T)$
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 8.5: Evaluation values for solutions to the Efficient Node Use Problem for small values of n

Experiment 8.2 *Write or obtain software for an evolutionary algorithm to solve the Efficient Node Use Problem. Use a population of 100 parse trees with n nodes. Use tournament selection with tournament size 4 as your model of evolution. If a child has more than n nodes after crossover, apply the chop operator until it has n or fewer nodes. Mutate each child with probability* 0.4. *Fitness of a tree T is simply $\epsilon(T)$.*

A tree T on n nodes is optimal *if $\epsilon(T)$ is as large as possible given n. The table given in Figure 8.5 contains the numbers that optimal trees evaluate to for $1 \leq n \leq 27$. For n=12, 13, 14, 15, and 16, run the evolutionary algorithm on a population until it finds an optimal tree or for 500 generations. Run 100 populations. Plot the fraction of successful populations as a function of generations of evolution. Also, note the fraction of initial populations that contained an optimal tree and the fraction that failed to find an optimal tree. In your write up, explain what happened and why. Is the probability of finding an optimal tree within 500 generations a decreasing function of the number of nodes in the tree?*

For use with experiments in Section 8.3, write the code for this experiment so that you can optionally save your population of parse trees in a file.

If your version of Experiment 8.2 worked properly, there should be a strange relationship between the number of nodes and the time-to-solution. In this case "strange" means "not monotone." Looking at the table given in Figure 8.5 can give you a clue to what is happening.

Our next experiment will break new ground. The point of this experiment is to give you practice at coming up with fitness functions for a difficult problem. Consider the Minimal Description Problem: given an integer k, find the PORS parse tree with the smallest number of nodes that evaluates to k. Problem 8.21 tells us we can restrict our population to trees with at most $2k + 1$ nodes. The difficulty comes in figuring out which trees with $2k + 1$ or fewer nodes are "close" to computing k. Examine Figure 8.6. It gives solutions to the

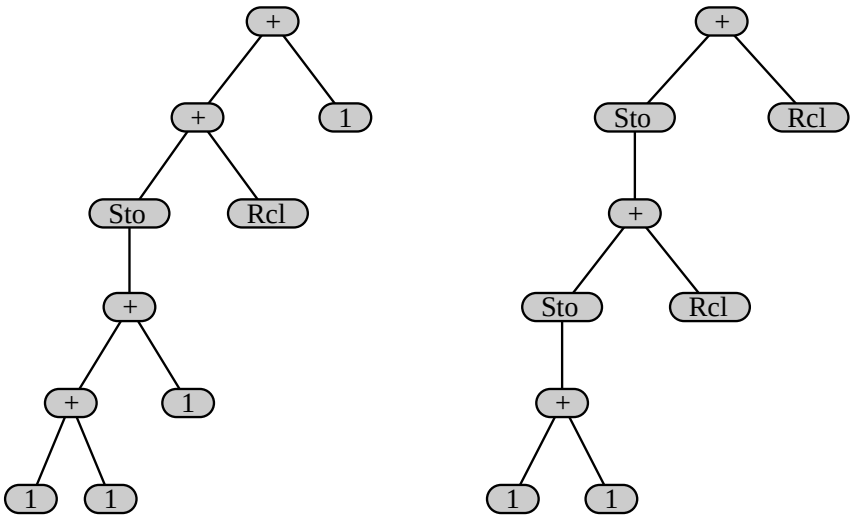


Figure 8.6: Minimal trees for computing 7 and 8

Minimal Description Problem for $k = 7, 8$. The two trees do not share much in the way of structure. The minimal description trees for $k = 11, 12$ are even less alike.

If minimal-sized trees that compute numbers differing by one are not structurally similar, then minimizing $|k - \epsilon(T)|$ is probably not a good fitness function for the Minimal Description Problem. What other choices are there? Are there any other senses in which two numbers, a and b, can be close? One obvious source of closeness is the divisibility relation. This yields several candidates: the greatest common divisor of a and b; the integer division of a by b or b by a; or the remainder of a divided by b or of b divided by a. These could all profitably appear in a definition of "close." It is possible that lexical products or numerical averages of these divisibility relations might serve as parts of fitness functions for the Minimal Description Problem.

In addition, to solve the Minimal Description Problem we want to minimize the number of nodes in the tree. This makes the Minimal Description Problem an example of *multicriteria optimization* with the multiple criteria being computation of numbers close to k and minimization of the number of nodes used to do so. We will call a PORS parse tree that evaluates to k with the fewest number of nodes a *minimal description tree* for k.

For those of you who have not taken a course in discrete mathematics, we will review the method for computing the greatest common divisor (GDC) of two numbers. Suppose we are given integers a and b. Perform the following computations:

$$x_0 \leftarrow a$$
$$x_1 \leftarrow b$$

k	$\|T\|_{min}$	k	$\|T\|_{min}$	k	$\|T\|_{min}$
1	1	10	11	19	15
2	3	11	13	20	14
3	5	12	11	21	15
4	6	13	13	22	16
5	8	14	13	23	18
6	8	15	13	24	14
7	1	16	12	25	16
8	9	17	14	26	16
9	10	18	13	27	15

Figure 8.7: The minimal number of nodes in a tree describing k, for small values of k

$$x_2 \leftarrow x_0 \; mod \; x_1$$
$$x_3 \leftarrow x_1 \; mod \; x_2$$
$$x_4 \leftarrow x_2 \; mod \; x_3$$
$$\ldots$$

until $x_n = 0$ for the first time. At that point $x_{n-1} = GCD(a, b)$.

Experiment 8.3 *Before writing any code, write a short essay on what you think will make a good fitness function for the Minimal Description Problem. Using your fitness function, rewrite the code from Experiment 8.2 to evolve parse trees with no more than* $2k + 1$ *nodes (apply the Chop operation to maintain this). Test your code on finding minimal description trees for* $k = 8, 10, 13,$ *and* 18 *with the same number of runs and reporting techniques as in Experiment 8.2. A table of sizes of answers for the Minimal Description Problem appears in Figure 8.7. Once you have done this testing, revise your fitness function and try again. If you find a killer fitness function, try it out on other numbers.*

Things to think about while doing this experiment include the following. What, if any, is the connection between the Efficient Node Use Problem and the Minimal Description Problem? Is the factorization of a number an important thing to know when trying to improve performance of your algorithm on that number?

If this experiment is being done by a whole class, you should work in teams: share data and ideas, and comment on one another's results in your write ups.

In this section, we have developed the basic framework of PORS and presented the two main problems. In the next section, we will explore seeding the population and test its effect on performance of evolutionary algorithms for both problems.

Problems

Problem 8.15 *For each of the following parse trees in Lisp-like notation, compute* $\epsilon(T)$.

a) **(Sto (+ 1 (+ 1 (Sto 1))))**

b) **(Sto (+ Rcl (Sto (Sto (+ Rcl (Sto 1))))))**

c) **(+ (+ (Sto (Sto 1)) 1) (+ Rcl (Sto (+ Rcl Rcl))))**

d) **(+ (+ (Sto (Sto (Sto Rcl))) (Sto 1)) (+ 1 (+ 1 1)))**

e) **(Sto (Sto (+ (+ 1 Rcl) 1)))**

f) **(+ (Sto Rcl) (+ Rcl (Sto Rcl)))**

g) **(Sto (Sto (+ (Sto (+ (Sto 1) (+ 1 Rcl))) (+ Rcl Rcl))))**

h) **(+ (+ (Sto 1) (Sto (+ Rcl 1))) (Sto (+ Rcl Rcl)))**

i) **(+ (Sto (Sto (+ (+ (Sto 1) 1) (Sto 1)))) (Sto Rcl))**

j) **(+ (+ (Sto (+ (Sto 1) 1)) (Sto (+ 1 Rcl))) (Sto Rcl))**

k) **(+ (Sto 1) (+ (Sto (+ 1 (+ 1 Rcl))) (+ Rcl Rcl)))**

l) **(+ (+ (Sto (+ 1 1)) (+ (Sto (+ 1 Rcl)) Rcl)) Rcl)**

Problem 8.16 *Prove that the minimal description of* 2^n *requires no more than* $3n$ *nodes.*

Problem 8.17 *Prove that the minimal description of* 3^n *requires no more than* $5n$ *nodes.*

Problem 8.18 *Let* $f(n) = Max\{\epsilon(T) : T \text{ is a PORS parse tree with } n \text{ nodes}\}$. *Prove that* $f(n)$ *is strictly increasing save that* $f(6) = f(7)$, *and* $f(1) = f(2)$.

Problem 8.19 *Prove that for* $n \geq 6$, *all solutions to the Efficient Node Use Problem on* n *nodes contain* **Sto** *instructions, but for odd* n *less than 6, solutions to the Efficient Node Use Problem do not contain* **Sto** *instructions. Explain why* $n = 2, 4$ *are pathological cases.*

Problem 8.20 Essay. *Consider the following sets of numbers: prime numbers, factorials, powers of two, and all numbers. Rank them from most to least difficult, on average, for the*

Minimal Description Problem, and explain your ranking. Assume the comparison is made between the sets by comparing members of similar magnitude.

Problem 8.21 *We say a PORS parse tree* describes *a number if it evaluates to that number. Prove that the description of a number k by a PORS parse tree requires at most $2k+1$ nodes.*

In the next several problems we will let $g(k)$ be the number of nodes in a minimal description tree for the number k.

Problem 8.22 *Show that $g(k+1) \leq g(k)+2$ and give one example of a k where the bound is tight.*

Problem 8.23 *Show that $g(2k) \leq g(k)+3$ and give one example where this bound is tight.*

Problem 8.24 *Show that $g(k)$ admits a logarithmic upper bound. This can be done constructively.*

Problem 8.25 Essay. *Explain the connections between the Efficient Node Use Problem and the Minimal Description Problem. What information does one give you about the other?*

Problem 8.26 *A* subroutine *in the PORS language is a connected subset of a tree with one link coming in and one going out. Give subroutines that multiply their input by 2, 3, and 5. Advanced students should give such subroutines with a minimal number of nodes.*

Problem 8.27 *The chart of answers given in Experiment 8.2 strongly suggests a closed form for the maximum number that can be generated by an n-node parse tree. Find that closed form.*

Problem 8.28 Short Essay. *Is the evolutionary algorithm in Experiment 8.2 doing an optimization?*

8.3 Seeding Populations

In this section, we want to explore the effects of a number of types of population seeding. Population seeding consists of placing creatures in the initial population which you feel will help the population to do what you want. The ultimate in population seeding is to place the answer to your problem in the original population. Often this is done in an algorithm using niche specialization in order to find variations on the solution or new types of solutions. We will explore two less ambitious types of seeding. In the first, we will used evolved creatures from one instance of the Efficient Node Use Problem as the starting population for another. In the second, we will try and apply common sense to the generation of the initial population.

Experiment 8.4 *Using the software from Experiment 8.2, evolve 10 populations of parse trees that contain solutions for the Efficient Node Use Problem for* $n = 8, 9,$ *and* 10 *respectively. For each of these 10 populations, use that population 10 times (with different random number seeds) as the initial population, as in Experiment 8.2, but with* $n = 25, 26,$ *and* 27*. First of all, did seeding help? Did the populations find solutions faster than you would have expected based on your experience so far? Second, in addition to the fraction of success per time graph, make a* 3×3 *matrix showing which populations were helped the most by which seed populations, as measured by fraction of successes after 500 generations.*

At this point, we will try a different population seeding technique. This technique rests on the assumption that a very simple statistical model of PORS parse trees in a successful population of trees with n nodes contains useful information about the trees that generalizes to other values of n.

Let us review how PORS parse trees are generated. A tree with one node is a **1** or a **Rcl** with equal probability. A tree with two nodes is a **Sto** which has a one node tree as its argument. A tree on three or more nodes is, with equal probability, a $+$ with the remaining nodes divided into nonzero parts between its arguments or a **Sto** with a tree on the remaining nodes as its argument.

The probability of finding any given parse tree is quite likely different in an evolved population containing optimal trees as compared to an initial random population. The basic plan is to use a statistical model to generate an initial population with statistics close to those in a successful population and hope this speeds up evolution. In order to gain any advantage, we must apply statistical models derived from populations of trees with a smaller number of nodes to generating initial populations of trees with a larger number of nodes. Our statistical model will look at the probability, for each operation, of each type of argument within trees and bias tree generation with those probabilities. We will divide the experiment into two parts.

Experiment 8.5 *Create or obtain software for a new routine to add to your library of PORS parse tree routines: given a parse tree, compute the number of times the left and right arguments of a* $+$ *and the descendants of a* **Sto** *are, respectively, a* $+$*,* **Sto***,* **Rcl***, or* **1***. Using the software from Experiment 8.2, evolve 10 populations of parse trees that contain solutions to the Efficient Node Use Problem on* $n = 12$ *nodes. Apply your new routine to compute, for the entire population, the fractions of left arguments of* $+$*, right arguments of* $+$*, and arguments of* **Sto** *that are, respectively,* $+$*,* **Sto***,* **Rcl***, or* **1***. Generate 1000 random trees in the usual manner and compute these same fractions. Are they different? Try and explain why.*

Think about what the results of Experiment 8.5 suggest about useful restrictions on the generation of initial populations of PORS parse trees. Hold that thought until after we have done the second half of the statistical modeling experiment.

Experiment 8.6 *Modify the random parse tree generation routine used to generate initial populations (but not the one used during mutation) so that nodes appear as arguments with probability equal to the corresponding fractions generated in Experiment 8.5.*

This will require a little work. For example, you will need 4 probabilities for the left argument of a $+$*,* P_+*,* P_{Sto}*,* P_1*, and* P_{Rcl}*. But, a one node tree that is to be the left argument of a* $+$ *is either a* **1** *or a* **Rcl***;* P_+ *and* P_{Sto} *are 0 for technical reasons when generating a one node tree. That means, in this situation, you use the dependent probabilities* $P_1^* = \frac{P_1}{P_1+P_{Rcl}}$ *and* $P_{Rcl}^* = \frac{P_{Rcl}}{P_1+P_{Rcl}}$ *instead of* P_1 *and* P_{Rcl}*. These are the probabilities of* **1** *and* **Rcl** given that *we know we are only choosing a* **1** *or a* **Rcl***.*

The general principle for generating parse trees with a statistical model is as follows. When all the probabilities we have are for events that are technically allowed, we use them in unmodified form. Otherwise we use the probabilities of things that are currently allowed divided by the sum of things currently allowed. This is exactly the notion of dependent probability that appears in statistics books.

Using your new initial tree generator, rerun Experiment 8.2 for $n = 15, 16,$ *and* 17*. For each of these n, what effect does the new generation method have on: speed of solution, fraction of runs with a solution in the initial population, and fraction of runs that fail to converge? Explain your results.*

Look again at your results for Experiment 8.5. One clear piece of good sense implicit in the statistics is that a **Sto** should not have another **Sto** as its argument. This is a cumbersome way to get this piece of wisdom, but there it is. (Perhaps you can find other bits of wisdom somehow encoded in the probabilities.) In the next experiment, we will begin using common sense to generate better initial populations by simply requiring that **Sto** operations have as their arguments things other than **Sto**.

Definition 8.1 *Let* $\mathcal{T}_s$ *be the set of PORS parse trees in which no* **Sto** *operation has a* **Sto** *as an argument.*

Experiment 8.7 *Add to your library of PORS parse tree routines a routine that can generate random trees in* $\mathcal{T}_s$*. Using this routine for generating the initial population (but not for mutation), redo Experiment 8.2 and compare the results with the results obtained in Experiment 8.2.*

If we comb through the populations evolved for the Efficient Node Use Problem, another property of optimal trees emerges. When evaluating a parse tree, the instructions are executed in some order. In particular, if a tree has **Sto** instructions, then it has a *first* **Sto** instruction. This execution order also orders the terminals of a tree and so we may separate the terminals into those that execute before the first **Sto** and those that execute after the first **Sto**. It is a property of optimal trees that all terminals executed before the first store are **1**s and all those executed afterwards are **Rcl**s (see Problem 8.36).

Definition 8.2 *Let $\mathcal{T}^*$ be the set of PORS parse trees with only* **1***s before the first* **Sto** *and only* **Rcl***s after. The terms "before" and "after" are relative to the order in which nodes are executed.*

The two conditions that are satisfied by trees in $\mathcal{T}_s$ and $\mathcal{T}^*$ don't interfere with one another so we will also give a name to their intersection.

Definition 8.3 *Let: $\mathcal{T}_s^* = \mathcal{T}_s \bigcap \mathcal{T}^*$.*

This gives us four classes of PORS parse trees, the class of all PORS parse trees and the three classes using common sense to restrict structure. We have already done a version of Experiment 8.2 for all trees and those in $\mathcal{T}_s$. The next experiment will complete the sweep.

Experiment 8.8 *Write or obtain new random tree generation routines to generate initial populations in $\mathcal{T}^*$ and $\mathcal{T}_s^*$. Run Experiment 8.7 again, doing the full experiment for both methods of generating the initial population. Compare the results with the results of Experiments 8.2 and 8.7.*

There is another possible improvement in the initial population. Notice that the argument of a **Sto** in an optimal tree is never a **Rcl** or a **1**. If we wanted to explore this class of trees, we could have another common-sense class of trees called $\mathcal{T}_r$ and also $\mathcal{T}_r^*$, $\mathcal{T}_{rs}$, and $\mathcal{T}_{rs}^*$. Exploration of the various r-types of trees would require quite a bit of fiddling and wouldn't add much to the development of GP-techniques. We will leave exploration of the r classes of trees to those intrigued by the mathematical theory of PORS, and instead dredge out several old friends from earlier chapters and test them in the PORS environment.

Problems

Problem 8.29 *Take the parse trees shown in Figure 8.6, copy them, and number the nodes in the order they are executed.*

Problem 8.30 Essay. *Does the answer to Problem 8.27 suggest an explanation for the 3×3 matrix in Experiment 8.4? Explain your answer.*

Problem 8.31 *How many PORS parse trees are there with n nodes?*

Problem 8.32 *How many PORS parse trees with n nodes are there in $\mathcal{T}_s$?*

Problem 8.33 *How many PORS parse trees are there with n nodes in $\mathcal{T}^*$?*

Problem 8.34 *How many PORS parse trees are there with n nodes in $\mathcal{T}_s^*$?*

Problem 8.35 *Of the three special classes of parse trees, $\mathcal{T}_s, \mathcal{T}^*$, and $\mathcal{T}_s^*$, which can be generated by simply carefully choosing the probabilities of a statistical model like the one used in Experiment 8.6?*

Problem 8.36 *Prove that any optimal tree (a solution to the Efficient Node Use Problem) is in $\mathcal{T}_s^*$.*

Problem 8.37 *Prove that if an optimal tree with n nodes contains a* **Sto** *instruction, then there exists an optimal tree with n nodes for which the left argument of the root node is a tree whose root is a* **Sto***.*

Problem 8.38 Essay. *In experiments 8.5 and 8.6, we split apart the problem of finding a statistical model for use in generating an initial population and running an algorithm that used that statistical model to generate its initial population. Describe in detail an evolutionary algorithm that would build a statistical model for one n and then start over with a larger n all in one environment. Conceivably this process could be done in a loop to solve both problems for successive n and refine the statistical model as n grew. In terms of the mathematics of PORS for the Efficient Node Use Problem, explain which small n might give good statistics for larger n.*

Problem 8.39 *Classify and count solutions to the Efficient Node Use Problem for $n \leq 27$. Figure 8.5 will help you to identify such solutions.*

Problem 8.40 Essay. *For the Efficient Node Use Problem, $n = 15$ yields a fitness function with an odious local optimum. What is it and why?*

8.4 Applying Advanced Techniques to PORS

In this section, we will apply various techniques from earlier chapters to the PORS environment and produce a few new ones. It is important to keep in mind that, except for certain very odd people like theoretical computer scientists and mathematicians, the various PORS problems are not themselves intrinsically interesting. PORS is intended as a very simple test environment for genetic programming, akin to the string evolver as it was used in Chapter 2.

The form of solutions to the Efficient Node Use Problem suggests that optimal trees must have relatively high depth. This gives us a natural place to test an idea from Chapter 5, lexical products of fitness functions.

Experiment 8.9 *Modify the software from Experiment 8.2 so that the fitness function is $\epsilon(t)$ lex depth(T) with ϵ dominant. Do the same data runs and reports. Does using the lexical product fitness function improve performance?*

A technique for improving optimization that we treated in Chapter 3 is niche specialization. For PORS parse trees, range niche specialization is a more natural choice than domain niche specialization (see Problem 8.45). The fitness function of the Efficient Node Use Problem produces natural numbers. These natural numbers are spaced out rather strangely as we saw in Experiment 8.3. This indicates that we should say two PORS parse trees are similar enough to decrease one another's fitness if they produce exactly the same value. In the terms used in Chapter 3, the similarity radius is zero. To do range niche specialization, we also need a penalty function. In the next experiment, we will compare a couple of penalty functions as well as assessing if range niche specialization helps at all.

Experiment 8.10 *If you need to, review Section 3.3 until you remember how range niche specialization works. Now modify the software from Experiment 8.2 to operate with range niche specialization for the penalty functions:*

(i) $q(m) = (m+3)/4$, *and*

(ii) $q(m) = \begin{cases} 1 & m \leq 4 \\ m/4 & otherwise \end{cases}$.

Do the same runs and make the same report as in Experiment 8.2. Compare the results. Which, if either, penalty function helps more? Do they help at all? Is it worth doing range niche specialization for the Efficient Node Use Problem?

It may be that niche specialization is more helpful for minimal description than efficient node use. The next experiment tests this conjecture.

Experiment 8.11 *Modify the software from Experiment 8.3 to operate with range niche specialization for the penalty functions:*

(i) $q(m) = (m+3)/4$,

(ii) $q(m) = \begin{cases} 1 & m \leq 4 \\ m/4 & otherwise \end{cases}$, *and*

(iii) $q(m) = \sqrt{m}$.

Do the same runs and make the same report as in Experiment 8.3. Compare the results. Which penalty function, if any, helps more? Do they help at all? Is it worth doing range niche specialization for the Minimal Description Problem?

So far each evolutionary algorithm we have run on a PORS problem has used tournament selection with tournament size 4. It might be interesting to explore some other models of evolution. Of the two PORS problems, the Efficient Node Use Problem and the Minimal

Description Problem, we have a much better handle on the Efficient Node Use Problem. It should be clear from earlier experiments that there is a large local optimum in the efficient node use fitness function when n is a multiple of 3. Keeping this in mind, do the following experiment.

Experiment 8.12 *Modify the software from Experiment 8.2 so that it can use other models of evolution. For all 8 possible combinations of*

Roulette selection

Rank selection

with

Random replacement

Random elite replacement

Absolute fitness replacement

Local elite replacement

do the same set of runs and report the same data as in Experiment 8.2. Compare the outcome of this experiment with that of Experiment 8.2. Which model of evolution works best for which values of n?

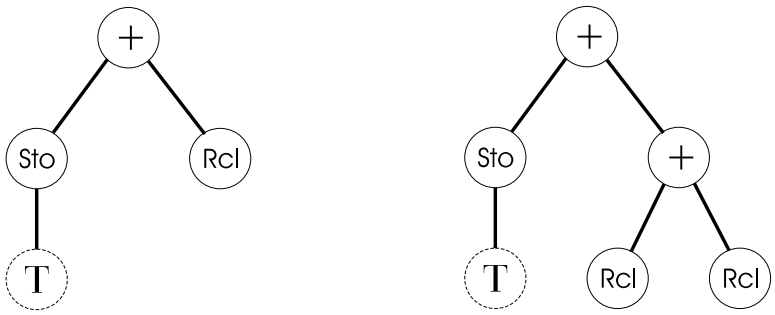


Figure 8.8: Subroutines for multiplying by 2 and 3

In Problem 8.26, the notion of subroutines is casually brought up. Recall that a *subroutine* in the PORS language is a connected subset of a tree with one link coming in and one going out. Figure 8.8 shows you subroutines for multiplying the output of any tree T by 2 or 3. (Alert readers will note this answers the two easier parts of Problem 8.26; think of this as a reward for students who read, or at least flip through, the text.)

So far our treatment of the Minimal Description Problem has been vague and left all the work to you. The subroutines in Figure 8.8 suggest a way of getting very small trees for numbers of the form 2^x3^y. In general, minimal subroutines for multiplying by m may be very good things to have in a population of parse trees being evolved to find a minimal description of a multiple of m. The next two experiments will put the Minimal Description Problem on a firmer footing. The first, somewhat bizarrely, should be called a POR experiment.

Experiment 8.13 *This experiment will require a revision of the standard PORS routines. In normal PORS, the memory is initialized with a 0 and the set of terminals is* {**1**,**Rcl**}. *Create a new set of parse tree manipulation routines from the PORS routines in which the external memory is initialized to 1, not 0, and all terminals are* **Rcl**. *Rebuild the software from Experiment 8.3 to use the new routines. Use the software to find minimal description trees in the POR language for k =2, 3, 5, 7, and 11. If you have not already done so, do Problem 8.42. Report different variations of the minimal description tree for each k.*

Once we have these POR parse trees that form minimal descriptions of various numbers (and have completed Problem 8.42), we have a source of subroutines that may help with the Minimal Description Problem (or cause us to dive into a local optimum, see Problem 8.44). In any case, the subroutines evolved in Experiment 8.13 may give us an effective way of seeding the population for the standard PORS Minimal Description Problem.

Experiment 8.14 *Add to your suite of PORS parse tree routines a routine that takes as arguments a maximum number of nodes and a tree of the type located in Experiment 8.13 for computing k. It should convert the tree into a subroutine for multiplying by k and then concatenate copies of the subroutine together with a one-node tree consisting of the terminal* **1** *so as to obtain a tree that computes k^m in the PORS environment, with m as large as possible. The bound on m is implied by having only n nodes available.*

Rebuild your software from Experiment 8.3 to operate with a seeded population. For each prime factor p of k, incorporate trees of the form p^m (m chosen to yield the largest power of p dividing k) as described above into the population (in close to equal numbers for all prime factors of k).

With the new software do the same type of runs and report the same things as in Experiment 8.3, but for k = 8, 10, 18, 21, 27, 35, and 63. Does the population seeding help at all? You may want to try revising your fitness function.

Problems

Problem 8.41 Essay. *Explain, to the best of your ability,* why *solutions to the Efficient Node Use Problem have tree depth much higher than random trees. Would you expect solutions to the Minimal Description Problem to share this property?*

Problem 8.42 *Read the description of Experiment 8.13. Prove that a minimal description tree for k evolved in the POR environment has the same size as a minimal subroutine in PORS for multiplying by k by constructing one from the other.*

Problem 8.43 *Compare and contrast the fitness function used in Experiment 8.2 with that used in Experiment 3.18. In what ways are they similar? Would Alife techniques that enhanced one tend to enhance the other? When or when not?*

Problem 8.44 *Prove or disprove: a minimal description tree for k and a minimum size subroutine for multiplying by k always have the same number of nodes.*

Problem 8.45 Essay. *In this section we explored the use of range niche specialization in the PORS environment. This is not difficult because the range of PORS parse trees is the natural numbers where there is a very natural measure of similarity to use. In order to do domain niche specialization we would need a way of telling if two PORS parse trees are close together. One possibility is simply to test and see if they are identical. Give another, with details, and compute the similarities of several example trees.*

Problem 8.46 Essay. *Clearly, if we find a tree on n nodes that evaluates to k then a minimal description tree for k has at most n nodes. Defend or attack the following proposition logically. While evolving a population to solve the Minimal Description Problem for a number k we should apply the chop operation to any tree with more nodes than the smallest tree we have found so far that evaluates to k.*