BBN Technical Report #7866: Strongly Typed Genetic Programming

David J. Montana
Bolt Beranek and Newman, Inc.
10 Moulton Street
Cambridge, MA 02138

May 7, 1993

Abstract

Genetic programming is a powerful method for automatically generating computer programs via the process of natural selection [Koza 92]. However, it has the limitation known as "closure", i.e. that all the variables, constants, arguments for functions, and values returned from functions must be of the same data type. To correct this deficiency, we introduce a variation of genetic programming called "strongly typed" genetic programming (STGP). In STGP, variables, constants, arguments, and returned values can be of any data type with the provision that the data type for each such value be specified beforehand. This allows the initialization process and the genetic operators to only generate parse trees such that the arguments of each function in each tree have the required types. An extension to STGP which makes it easier to use is the concept of generic functions, which are not true strongly typed functions but rather templates for classes of such functions. To illustrate STGP, we present three examples involving vector and matrix manipulation: (1) a basis representation problem (which can be constructed to be deceptive by any reasonable definition of "deception"), (2) the n-dimensional least-squares regression problem, and (3) preliminary work on the Kalman filter.

1 Introduction

Genetic programming is a method of automatically generating computer programs to perform specified tasks [Koza 92]. It uses a genetic algorithm to search through a space of possible computer programs for one which is nearly optimal in its ability to perform a particular task. In Section 1.1 we give a very brief overview of genetic algorithms (the unindoctrinated reader is referred to [Goldberg 89] as an introduction). In Section 1.2 we discuss how genetic programming differs from a standard genetic algorithm.

1.1 Genetic Algorithms

Genetic algorithms are a class of algorithms for optimization and learning based on the principles of natural evolution. They have been shown to be capable of finding nearly global optima in large and complex spaces in a relatively short time. Acccording to [Davis 87], a genetic algorithm has five basic components:

$$(16, -4, 2, 6, 13, -11) \xrightarrow{\text{mutation}} (16, -4, 2, 23, 13, -11)$$

$$\begin{matrix}(16, -4, 2, 6, 13, -11)\\(-7, 8, 14, -14, -15, -9)\end{matrix} \xrightarrow{\text{crossover}} (-7, 8, 2, 6, 13, -9)$$

Figure 1: Mutation and crossover for string-based genetic algorithms.

1. A representation scheme provides a way to code possible solutions to a problem in a form that is readily manipulable by the genetic operators. The traditional, and still most common, representation is as a fixed-length binary string, but any representation is acceptable as long as there are appropriate genetic operators defined.

2. An evaluation function (or fitness function) assigns a numerical score to any element of the search space. For function optimization problems, the evaluation function is the function that is being optimized.

3. An initialization procedure provides a way to randomly select the individuals (i.e., search points) which constitute the initial population. For a fixed-length string representation, the usual approach is to select each field of each string randomly from its possible values.

4. A set of genetic operators provides a way to use the information from one or more search points to stochastically generate new search points. The two standard genetic operators are mutation and crossover. Mutation takes a single "parent" and produces a "child" which has the same information as the parent in all but a small number of locations, where new information is randomly generated. Crossover takes two parents and produces one or two children whose information is some combination of the information from the parents. Figure 1 shows examples of mutation and crossover acting on fixed-length real-valued strings.

5. A variety of parameter settings determine the run-time characteristics of the genetic algorithm. Such parameters include POPULATION-SIZE, GENERATION-SIZE (when the generation size equals the population size there is full generational replacement and when the generation size is one it is a steady-state genetic algorithm), and the operator selection probabilities.

Given these five components, a genetic algorithm operates according to the following steps (and as pictured in Figure 2):

1. Using the initialization process, generate an initial population of size POPULATION-SIZE, and evaluate these individuals.

2. Generate a new generation of size GENERATION-SIZE via the process of reproduction. Creation of a single new inidividual occurs as follows:

   (a) Randomly select an operator according to the operator selection probabilities.

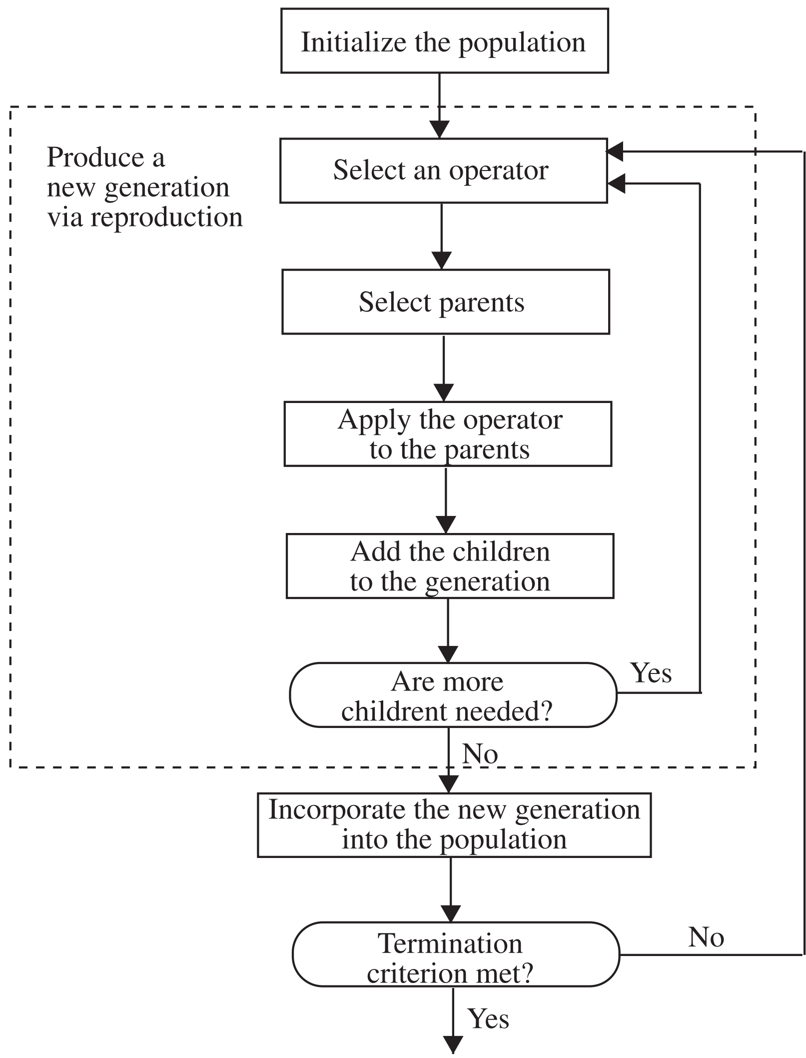

Figure 2: Control flow for a genetic algorithm.

   (b) Randomly select parents from the current population with the probability of selecting a particular individual monotonically increasing with the fitness of the individual. (The number of parents to be selected is determined by the operator.)

   (c) Apply the operator to the parents to create one or more children.

3. Remove the GENERATION-SIZE worst member of the current population and replace them with the new generation.

4. If a termination criterion is not met, repeat from step (2).

```
begin procedure
    x := ( x + 3 ) * ( 4 - y );
    y := x + y - 1;
end procedure;
```

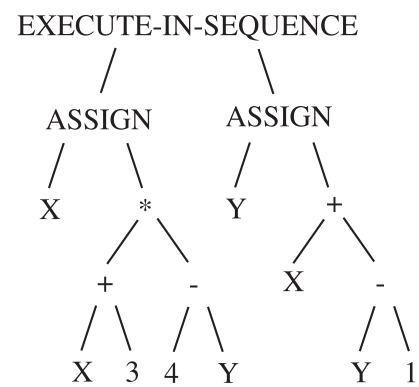

Figure 3: A subroutine and an equivalent parse tree.

1.2 Genetic Programming

We now describe the five components of the type of genetic algorithm used for genetic programming:

(1) **Representation -** For genetic programming, computer programs are represented as parse trees. A parse tree is a tree whose nodes are procedures, functions, variables and constants. The subtrees of a node in a parse tree represent the arguments to the procedure or function of that node. (Since variable and constants take no arguments, their nodes can never have subtrees, i.e. they always are leaves.) Executing a parse tree means executing the root of the tree, which executes its children nodes as appropriate, and so on recursively.

Any subroutine can be represented as a parse tree. For example, the subroutine shown in Figure 3 is represented by the parse tree shown in Figure 3. Here, the EXECUTE-IN-SEQUENCE procedure just executes each of its arguements in order, while the ASSIGN procedure just executes its second argument and assigns its value to the variable specified by its first argument.

While the conversion from a subroutine to its parse tree is non-trivial in languages such as C, PASCAL and Ada, in the language LISP a subroutine essentially is its parse tree. Although we wish to downplay the association with LISP and have implemented our genetic programming code in C++, LISP does provide a compact way to express a parse tree in a linear fashion. Each node representing a variable or a constant is expressed as the name of the variable or value of the constant. Each node representing a function is expressed by a '(' followed by the function name followed by the expressions for each subtree in order followed by a ')'. A LISP expression for the parse tree of Figure 3 is

```
(EXECUTE-IN-SEQUENCE (ASSIGN X (* (+ X 3) (- 4 Y))) (ASSIGN Y (+ X (- Y 1))))
```

For genetic programming, the user defines all the possible functions, variables and constants that can be used as nodes in a parse tree. Variables, constants, and functions which take no arguments are the leaves of the possible parse trees and hence are called "terminals". Functions which do take arguments, and therefore are the branches of the possible parse trees, are called "non-terminals". The set of all terminals is called the "terminal set", and the set of all non-terminals is called the "non-terminal set".

[An aside on terminology: We use the term "non-terminal" to describe what Koza [92] calls a "function". This is because a terminal can be what standard computer science nomenclature would call a "function", i.e. a subroutine that returns a value.]

An important constraint on the user-defined terminals and non-terminals is called **closure**. Closure means that all these elements take arguments of a single data type (e.g., a scalar) and return values of this same data type. This implies that all elements return values that can be used as arguments for any element; hence, any element can be a child node in a parse tree for any other element without having conflicting data types. Koza [92] describes a way to relax this constraint of closure a little with the concept of "constrained syntactic structures". Here, data structures which have argument types or return types that are not standard can occur in a tree but only at a preassigned position. In Section 2, we discuss a way to fully relax the closure constraint.

The search space is the set of all parse trees which use only elements of the non-terminal set and terminal set and which are legal (i.e., have the right number of arguments for each function) and which are less than some maximum depth. This limit on the maximum depth is a parameter which keeps the search space finite and prevents trees from growing to an unmanageably large size.

(2) **Evaluation Function** - The evaluation function consists of executing the program defined by the parse tree and scoring how well the results of this execution match the desired results. The user must supply the function which assigns a numerical score to how well a set of derived results matches the desired results.

(3) **Initialization Procedure** - Koza [92] defines two different ways of generating a member of the initial population, the "full" method and the "grow" method. For a parse tree generated by the full method, the length along any path from the root to a leaf is the same nomatter which path is taken, i.e. the tree is of full depth along any path. Parse trees generated by the grow method need not satisfy this constraint. For both methods, each tree is generated recursively using the following algorithm described in pseudo-code:

```
Generate_Tree( max_depth, generation_method )
begin
    if max_depth = 1 then
        set the root of the tree to a randomly selected terminal;
    else if generation_method = full then
        set the root of the tree to a randomly selected non-terminal;
    else
        set the root to a randomly selected element which is either
            terminal or non-terminal;
    for each argument of the root, generate a subtree with the call
        Generate_Tree( max_depth - 1, generation_method );
end;
```

The standard approach of Koza to generating an initial population is called "ramped-half-and-half". It uses the full method to generate half the members and the grow method to generate the other half. The maximum depth is varied between two and MAX-INITIAL-TREE-DEPTH. This approach generates trees of all different shapes and sizes.

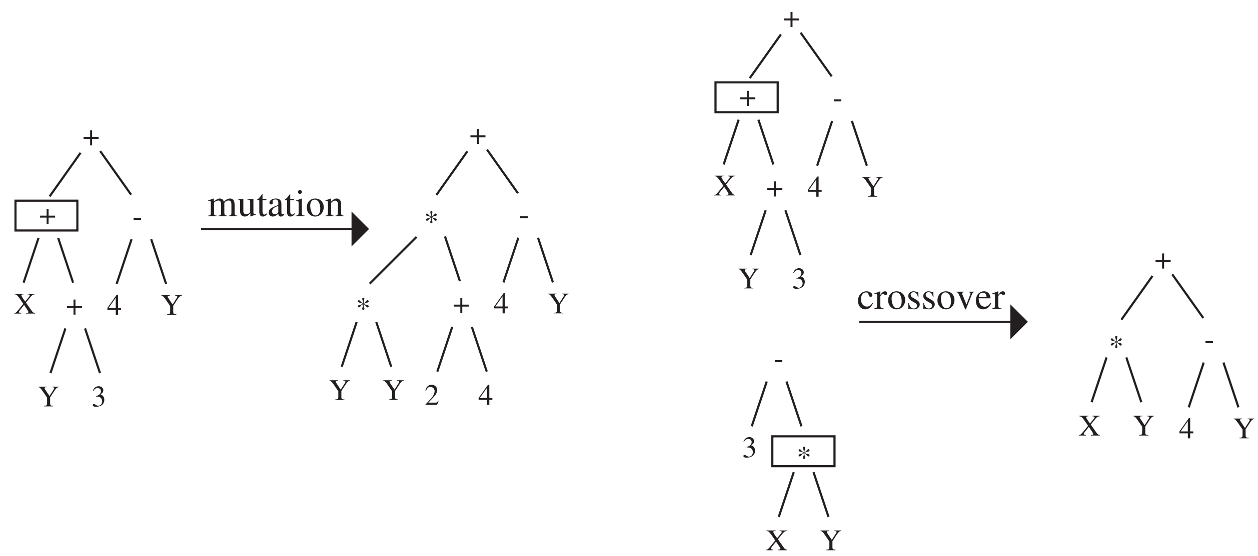

Figure 4: Mutation and crossover for genetic programming.

(4) **Genetic Operators -** Like a standard genetic algorithm, the two main genetic operators are mutation and crossover (although Koza [92] claims that mutation is generally unnecessary). However, because of the tree-based representation, these operators must work differently from the standard mutation and crossover. Mutation works as follows: (i) randomly select a node within the parent tree as the mutation point, (ii) generate a new tree of maximum depth MAX-MUTATION-TREE-DEPTH, (iii) replace the subtree rooted at the selected node with the generated tree, and (iv) if the maximum depth of the child is less than or equal to MAX-TREE-DEPTH, then use it. (If the maximum depth is greater than MAX-TREE-DEPTH, then we can either use the parent (as Koza does) or start again from scratch (as we do).) The mutation process is illustrated in Figure 4.

Crossover works as follows: (i) randomly select a node within each tree as crossover points, (ii) take the subtree rooted at the selected node in the second parent and use it to replace the subtree rooted at the selected node in the first parent to generate a child (and optionally do the reverse to obtain a second child), and (iii) use the child if its maximum depth is less than or equal to MAX-TREE-DEPTH. The crossover procedure is illustrated in Figure 4.

(5) **Parameters -** There are some parameters associated with genetic programming beyond those used with standard genetic algorithms. MAX-TREE-DEPTH is the maximum depth of any tree. MAX-INITIAL-TREE-DEPTH is the maximum depth of a tree which is part of the initial population. MAX-MUTATION-TREE-DEPTH is the maximum depth of a subtree which is generated by the mutation operator as the part of the child tree not in the parent tree.

2 Strongly Typed Genetic Programming (STGP)

We now discuss an extension of the basic genetic programming approach called strongly typed genetic programming (STGP). The key contribution of STGP is that it eliminates the closure constraint described above and hence allows functions which take arguments of any data type and return values of any data type. It does this by requiring that each function specify precisely the data types of its arguments and its returned values (i.e., that the functions be "strongly typed"). STGP can then ensure that all the parse trees it generates satisfy the constraint that the arguments to all functions are of the correct type.

Section 2.1 discusses the details of the extensions from basic genetic programming needed to ensure that all the argument types are correct. Section 2.2 describes a key concept for making STGP easier to use, generic functions, which are not true strongly typed functions but rather templates for classes of strongly typed functions.

2.1 The Basics

We now discuss in detail the changes from standard genetic programming for each genetic algorithm component:

(1) **Representation** - In STGP, unlike in standard genetic programming, each variable and constant has an assigned type. For example, the constants 1 and π have the type SCALAR, the variable $V1$ might have the type VECTOR-3 (indicating a three-dimensional vector), and the variable $M2$ might have the type MATRIX-2-2 (indicating a 2x2 matrix).

Furthermore, each function has a specified type for each argument and for the value it returns. For example, DOT-PRODUCT-3 takes two arguments of type VECTOR-3 and returns a value of type SCALAR; VECTOR-ADD-2 takes two arguments of type VECTOR-2 and returns a value of type VECTOR-2; MAT-VEC-MULT-4-3 takes one argument of type MATRIX-4-3 and one of type VECTOR-3 and returns a value of type VECTOR-4. (Note that below we will describe generic functions, which provide a way to define a single function, e.g. DOT-PRODUCT, instead of many functions which do essentially the same operation, e.g. DOT-PRODUCT-i for i different values.)

To handle multiple data types, the definition of what constitutes a legal parse tree has a few additional criteria beyond those required for standard genetic programming, which are: (i) the root node of the tree returns a value of the type required by the problem, and (ii) each non-root node returns a value of the type required by the parent node as an argument. These criteria for legal parse trees are illustrated by the following example:

Example 1 Consider a non-terminal set $\mathcal{N}$ = {DOT-PRODUCT-2, DOT-PRODUCT-3, VECTOR-ADD-2, VECTOR-ADD-3, SCALAR-VEC-MULT-2, SCALAR-VEC-MULT-3} and a terminal set $\mathcal{T}$ = {V1, V2, V3}, where V1 and V2 are variables of type VECTOR-3 and V3 is a variable of type VECTOR-2. Let the required return type be VECTOR-3. Then, Figure 5 shows an example of a legal tree. Figure 6 shows two examples of illegal trees, the left tree because its root returns the wrong type and the right tree because in three places the argument types do not match the return types.

(2) **Evaluation Function** - There are no changes to the evaluation function.

(3) **Initialization Procedure** - The one change to the initialization procedure is that, unlike in standard genetic programming, there are type-based restrictions on which element can be chosen at each node. One

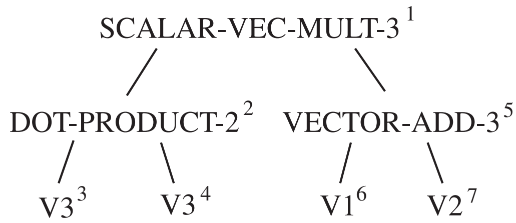

Figure 5: An example of a legal tree for return type VECTOR-3.

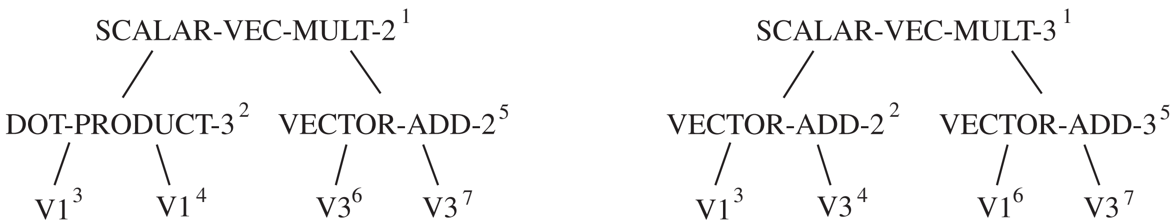

Figure 6: Two examples of illegal trees for return type VECTOR-3.

restriction is that the element chosen at that node must return the expected type (which for the root node is the expected return type for the tree and for any other node is the argument type for the parent node). A second restriction is that, when recursively selecting nodes, we cannot select an element which makes it impossible to select legal subtrees. (Note that if it is impossible to select any tree for the specified depth and generation method, then no tree is returned, and the initialization procedure proceeds to the next depth and generation method.) We discuss this second restriction in greater detail below but first give an example of this tree generation process.

Example 2 Consider using the full method to generate a tree of depth 3 returning type VECTOR-3 using the terminal and non-terminal sets of Example 1. We now give a detailed description of the decision process that would generate the tree in Figure 5. At point 1, it can choose either SCALAR-VEC-MULT-3 or VECTOR-ADD-3, and it chooses SCALAR-VEC-MULT-3. At point 2, it can choose either DOT-PRODUCT-2 or DOT-PRODUCT-3 and chooses DOT-PRODUCT-2. At points 3 and 4, it can only choose V3, and it does. At point 5, it can only choose VECTOR-ADD-3. (Note that there is no tree of depth 2 with SCALAR-VEC-MULT-3 at its root, and hence SCALAR-VEC-MULT-3 is not a legal choice even though it returns the right type.) At points 6 and 7, it can choose either V1 or V2 and chooses V1 for point 6 and V2 for point 7.

Regarding the second restriction, we observe that a non-terminal element can be the root of a tree of maximum depth i if and only if all of its argument types can be generated by trees of maximum depth $i-1$. To check this condition efficiently, we use "types possibilities tables", which we generate before generating the first tree. Such a table tells for each $i = 1,...,$MAX-INITIAL-TREE-DEPTH what are the possible return types for a tree of maximum depth i. There will be two different types possibilities tables, one for trees generated by the full method and one for the grow method. Example 4 below shows that these two tables are not necessarily the same. The following is the algorithm in pseudo-code for generating these tables.

```
-- the trees of depth 1 must be a single terminal element
loop for all elements of the terminal set
    if table_entry( 1 ) does not yet contain this element's type
        then add this element's type to table_entry( 1 );
end loop;

loop for i = 2 to MAX_INITIAL_TREE_DEPTH
    -- for the grow method trees of size i-1 are also valid trees of size i
    if using the grow method
        then add all the types from table_entry( i-1 ) to table_entry( i );
    loop for all elements of the non-terminal set
        if this element's argument types are all in table_entry( i-1 ) and
           table_entry( i ) does not contain this element's return type
            then add this element's return type to table_entry( i );
    end loop;
end loop;
```

Example 3 For the terminal and non-terminal sets of Example 1, the types possibilities tables for both the full and grow method are

```
table_entry( 1 ) = { VECTOR-2, VECTOR-3 }
table_entry( i ) = { VECTOR-2, VECTOR-3, SCALAR } for i > 1
```

Note that in Example 1, when choosing the node at point 5, we would have known that SCALAR-VEC-MULT-3 was illegal by seeing that SCALAR was not in the table entry for depth 1.

Example 4 Consider the case when $\mathcal{N}$ = {MAT-VEC-MULT-3-2, MAT-VEC-MULT-2-3, MATRIX-ADD-2-3, MATRIX-ADD-3-2} and $\mathcal{T}$ = {M1, M2, V1}, where M1 is of type MATRIX-2-3, M2 is of type MATRIX-3-2, and V1 is of type VECTOR-3. Then, the types possibilities tables for the grow method is

```
table_entry( 1 ) = { VECTOR-3, MATRIX-3-2, MATRIX-2-3 }
table_entry( i ) = { VECTOR-2, VECTOR-3, MATRIX-3-2, MATRIX-2-3 } for i > 1
```

and the types possiblities table for the full method is

```
table_entry( i ) = { VECTOR-3, MATRIX-3-2, MATRIX-2-3 } for i odd
table_entry( i ) = { VECTOR-2, MATRIX-3-2, MATRIX-2-3 } for i even
```

(4) **Genetic Operators** - The genetic operators, like the initial tree generator, must respect the enhanced legality constraints on the parse trees. Mutation uses the same algorithm employed by the initial tree generator to create a new subtree which returns the same type as the deleted subtree and which has internal

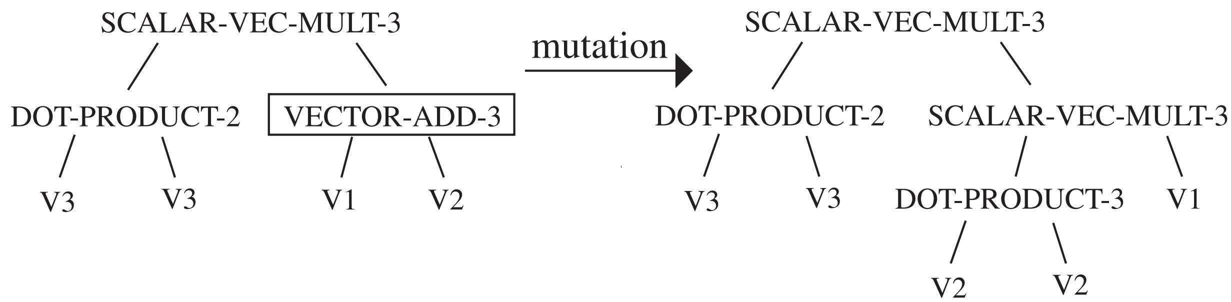

Figure 7: Mutation for STGP.

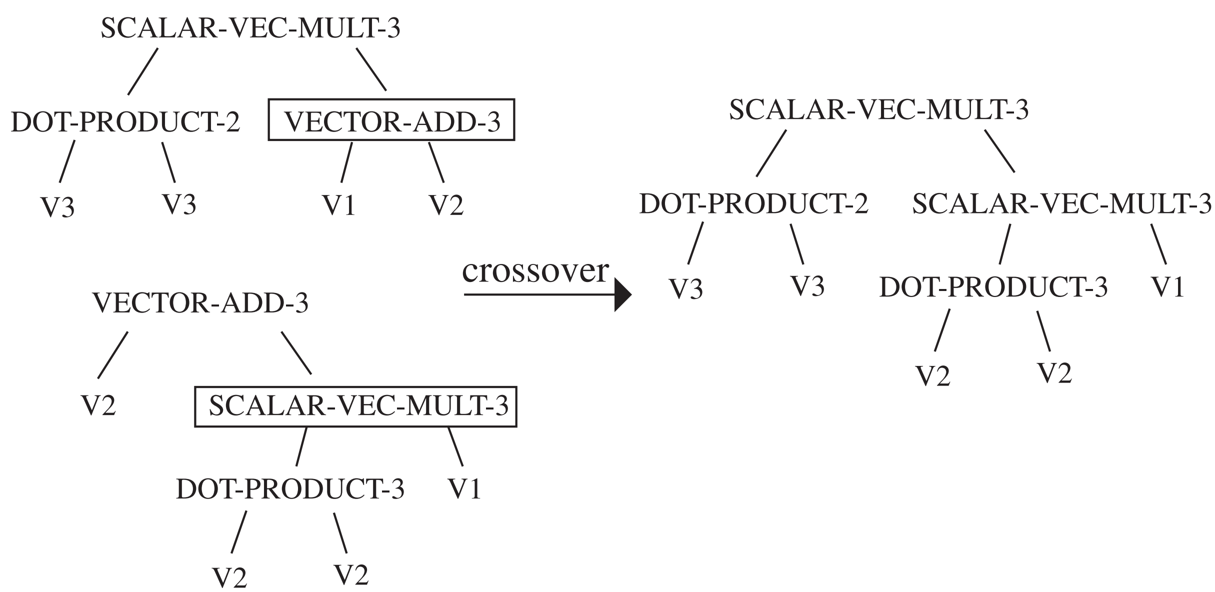

Figure 8: Crossover for STGP.

consistency between argument types and return types (see Figure 7). If it is impossible to generate such a tree, then the mutation operator returns either the parent or nothing.

Crossover now works as follows. The crossover point in the first parent is still selected randomly from all the nodes in the tree. However, the crossover point in the second parent must be selected so that the subtree returns the same type as the subtree from the first parent. Hence, the crossover point is selected randomly from all nodes satisfying this constraint (see Figure 8). If there is no such node, then the crossover operator returns either the parents or nothing.

(5) **Parameters -** There are no changes to the parameters.

2.2 Generic Functions

The examples above illustrate a major inconvenience of the basic STGP formulation, the need to specify multiple functions which perform the same operation on different types. For example, it is inconvenient to have to specify both DOT-PRODUCT-2 and DOT-PRODUCT-3 instead of a single function DOT-PRODUCT. To eliminate this inconvenience, we introduce the concept of a "generic function". A generic function is a function which can take a variety of different argument types and, in general, return values of a variety of different types. The only constraint is that for any particular set of argument types a generic function must return a value of a well-defined type. Specifying a set of argument types (and hence also the return type) for a generic function is called "instantiating" the generic function.

Some example of generic functions are DOT-PRODUCT, VECTOR-ADD, MAT-VEC-MULT, MATRIX-TRANSPOSE, MATRIX-INVERSE, and IF-THEN-ELSE. Legal sets of argument types for DOT-PRODUCT are all pairs (VECTOR-i, VECTOR-i) where i is any positive integer. DOT-PRODUCT always returns type SCALAR. For any set of argument types (VECTOR-i, VECTOR-i), ADD-VECTOR returns type VECTOR-i. For any set of argument types (MATRIX-i-j, VECTOR-j), MAT-VEC-MULT returns type VECTOR-i. For any argument of type MATRIX-m-n, MATRIX-TRANSPOSE returns type MATRIX-m-n. For any argument of type MATRIX-n-n, MATRIX-INVERSE returns type MATRIX-n-n. (Note that our version of MATRIX-INVERSE works analogously to Koza's protected divide function; if the input matrix is not invertible, then it returns the identity matrix of the appropriate size. A matrix is defined as non-invertible if the Gaussian elimination process needs to find the reciprocal of a number smaller than 10^{-6}.) For any set of three arguments the first of which is type BOOLEAN and the last two of which are the same type, IF-THEN-ELSE returns type the same as that of the last two arguments.

To be in a parse tree, a generic function must be instantiated. Once instantiated, an instance of a generic function keeps the same argument types even when passed from parent to child. Hence, an instantiated generic function acts exactly like a standard strongly typed function. A generic function gets instantiated during the process of generating parse trees (for either initialization or mutation). Note that there can be multiple instantiations of a generic function in a single parse tree.

Because generic functions act like standard strongly typed functions once instantiated, the only changes to the STGP algorithm needed to accomodate generic functions are for the tree generation procedure. There are three such changes required.

First, during the process of generating the types possibilities tables, recall that for standard non-terminal functions we needed just check that each of its argument types was in the table entry for depth $i-1$ in order to add its return type to the table entry for depth i. This does not work for generic functions because each generic function has a variety of different argument types and return types. For generic functions, this step is replaced with the following:

```
loop over all ways to combine the types from table_entry( i-1 ) into
      sets of argument types for the function
    if the set of arguments types is legal
       and the return type for this set of argument types is not in
             table_entry( i )
        then add the return type to table_entry( i );
```

```
end loop;
```

The second change is during the tree generation process. Recall that for standard functions, when deciding whether a particular function could be child to an existing node, we could independently check whether it returns the right type and whether its argument types can be generated. However, for generic functions we must replace these two tests with the following single test:

```
loop over all ways to combine the types from table_entry( i-1 ) into
      sets of argument types for the function
    if the set of arguments types is legal
       and the return type for this set of argument types is correct
        then return that this function is legal;
end loop;
return that this function is not legal;
```

The third change is also for the tree generation process. Note that there are two types of generic functions, ones whose argument types are fully determined by selection of their return types and ones whose argument types are not fully determined by their return types. We call the latter "generic functions with free arguments". Some examples of generic functions with free arguments are DOT-PRODUCT and MAT-VEC-MULT, while some examples of generic functions without free arguments are VECTOR-ADD and SCALAR-VEC-MULT. When we select a generic function with free arguments to be a node in a tree, its return type is determined by its parent node (or if it is at the root position, by the required tree type), but this does not fully specify its argument types. Therefore, to determine its arguments types and hence the return types of its children nodes, we must use the types possibilities table to determine all the possible sets of argument types which give rise to the determined return type (there must be at least one such set for this function to have been selected) and randomly select one of these sets.

Example 5 Using generic functions, we can rewrite the non-terminal set from Example 1 in a more compact form: $\mathcal{N}$ = {DOT-PRODUCT, VECTOR-ADD, SCALAR-VEC-MULT}. Recall that $\mathcal{T}$ = {V1, V2, V3}, where V1 and V2 are type VECTOR-3, and V3 is type VECTOR-2. The types possibilities tables are still as in Example 3. Figure 9 shows the equivalent of the tree in Figure 5. To generate the tree shown in Figure 9 as an example of a full tree of depth 3, we go through the following steps. At point 1, we can select either VECTOR-ADD or SCALAR-VEC-MULT, and we choose SCALAR-VEC-MULT. At point 2, we must select DOT-PRODUCT. Because DOT-PRODUCT has free arguments, we must select its argument types. Examining the types possibilities table, we see that the pairs (VECTOR-2, VECTOR-2) and (VECTOR-3, VECTOR-3) are both legal. We randomly select (VECTOR-2, VECTOR-2). Then, points 3 and 4 must be of type VECTOR-2 and hence must be V3. Point 5 must be VECTOR-ADD. (SCALAR-VEC-MULT is illegal because SCALAR is not in the types possibilities table entry for depth 1.) Points 6 and 7 can both be either V1 or V2, and we choose V1 for point 6 and V2 for point 7.

3 Examples

We now discuss three problems to which we have applied STGP.

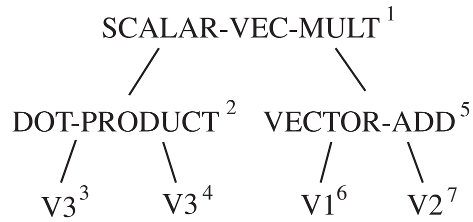

Figure 9: A legal tree using generic functions.

3.1 Basis Decomposition

Consider the following problem. Let $\mathcal{N}$ = {VECTOR-ADD} and $\mathcal{T}$ = {V1, V2}, where V1 and V2 are type VECTOR-3 and V1 = (2, 3, 4) and V2 = (3, 0, 0). Let the required return type for the tree be VECTOR-3 and the evaluation function be the distance of the returned vector from the vector (200, 120, 160) using the city-block metric (i.e., the sum of the absolute values of the differences of the components).

An optimal solution to this problem is any with 40 instances of V1 and 40 instances of V2 added together, and such a solution yields a distance of zero. This is a very simple problem which our genetic algorithm generally solves after evaluating 200-250 points.

To make this problem much harder, consider adding another vector V3 = (7, 5, 6) to the terminal set and keeping everything else the same. This seemingly minor change makes the problem "deceptive" by any reasonable definition of this term for a tree-based genetic algorithm. For genetic algorithms based on fixed-length strings, [Goldberg 87] defines a deceptive problem as one where the lower-order schema give information which lead the search away from the optimal solution. While there is as yet no equivalent formal definition for tree-based genetic algorithms, we note that at the early stages of the genetic algorithm, having many instances of V3 makes an individual appear to be more fit than other individuals. Hence, instances of V3 abound in the population. There are ways to obtain good but sub-optimal solutions with many instances of V3, and the population can easily find these solutions. However, once the population converges on these solutions, it is difficult to find the better solutions with less instances of V3. There is one solution with 22 V1's, 24 V2's and 12 V3's added together which yields a score of 6. To improve upon this requires leaping to a tree with 31 V1's, 32 V2's and 6 V3's, which has a score of 3. An even tougher leap is from this solution to the optimal one, which contains 40 V1's, 40 V2's and 0 V3's. Since early information leads to the wrong part of the space and since this part of the space is hard to leave once found, this problem should be considered deceptive.

3.2 n-Dimensional Least Squares Regression

The n-dimensional least squares regression problem can be stated as follows. For an mxn matrix A with $m > n$ and an m-vector B, find the n-vector X which minimizes the quantity $(AX - B)^2$. This problem is known to have the solution

$$X = (A^T A)^{-1} A^T B \tag{1}$$

where $(A^T A)^{-1} A^T$ is called the "pseudo-inverse" of A. Note that this is a generalization of the linear regression problem, given m pairs of data (x_i, y_i), find m and b such that the line $y = mx + b$ gives the best least-squares fit to the data. For this special case,

$$A = \begin{bmatrix} x_1 & 1 \\ \dots & \dots \\ x_m & 1 \end{bmatrix} \quad B = \begin{bmatrix} y_1 \\ \dots \\ y_m \end{bmatrix} \quad X = \begin{bmatrix} m \\ b \end{bmatrix} \tag{2}$$

To solve this problem using STGP, we defined a 20x3 matrix A and a 20-dimensional vector B each with randomly selected entries. We used the terminal set

$$\mathcal{T} = \{\mathrm{A}, \mathrm{B}\} \tag{3}$$

We used two different non-terminal sets

$$\mathcal{N}_1 = \{\text{MATRIX\_TRANSPOSE}, \text{MATRIX\_INVERSE}, \text{MAT\_VEC\_MULT}, \text{MAT\_MAT\_MULT}\} \tag{4}$$

$$\begin{aligned} \mathcal{N}_2 = \{&\text{MATRIX\_TRANSPOSE}, \text{MATRIX\_INVERSE}, \text{MAT\_VEC\_MULT}, \text{MAT\_MAT\_MULT}, \\ &\text{MATRIX\_ADD}, \text{MATRIX\_SUBTRACT}, \text{VECTOR\_ADD}, \text{VECTOR\_SUBTRACT}\} \end{aligned} \tag{5}$$

Note that $\mathcal{N}_1$ is the minimal non-terminal set needed to solve the problem.

For both non-terminal sets, the power of genetic algorithms is not necessary for finding the optimal solution. In fact, using $\mathcal{N}_1$ as the non-terminal set, in a typical run an initial population of 50 random individuals had 21 unique individuals which were equivalent to the optimal solution. (Note that the individuals in the population of our genetic algorithm are necessarily unique because we use a steady-state genetic algorithm which enforces uniqueness.) Using $\mathcal{N}_2$ as the non-terminal set, in a typical run an initial population of 500 random individuals had eight unique individuals which were equivalent to the optimal. Two optimal parse trees with the minimum number of nodes are:

```
(1) (MAT-VEC-MULT (MATRIX-INVERSE (MAT-MAT-MULT (MATRIX-TRANSPOSE A) A))
                  (MAT-VEC-MULT (MATRIX-TRANSPOSE A) B))
(2) (MAT-VEC-MULT (MAT-MAT-MULT
                        (MATRIX-INVERSE (MAT-MAT-MULT (MATRIX-TRANSPOSE A) A))
                        (MATRIX-TRANSPOSE A))
                  B)
```

An example of an optimal parse tree with more than the minumum nodes is

```
(MAT-VEC-MULT
   (MATRIX-TRANSPOSE (MATRIX-INVERSE (MAT-MAT-MULT (MATRIX-TRANSPOSE A) A)))
   (MAT-VEC-MULT (MATRIX-TRANSPOSE (MATRIX-TRANSPOSE (MATRIX-TRANSPOSE A))) B))
```

and there are many more such optimal parse trees of non-minimal size.

While this problem is too simple to stress the genetic algorithm, it does illustrate very nicely the advantage of STGP over standard genetic programming. Without the ability to handle vectors and matrices and the functions which manipulate them, standard genetic programming would have had to learn the structure inherent in the 80 pieces of data (60 entries for A and 20 entries for B) as well as learning operations equivalent to matrix inversion and transpose. This would turn a simple problem into a dauntingly difficult one.

3.3 The Kalman Filter

The Kalman filter is a popular method for tracking the state of a system with stochastic behavior using noisy measurements [Kalman 60]. A standard formulation of a Kalman filter is the following. Assume that the system follows the stochastic equation

$$\dot{\vec{x}} = A\vec{x} + B\vec{n}_1 \tag{6}$$

where $\vec{x}$ is an n-dimensional state vector, A is an nxn matrix, $\vec{n}_1$ is an m-dimensional noise vector, and B is an nxm matrix. We assume that the noise is Gaussian distributed with mean 0 and covariance the mxm matrix Q. Assume that we also make continuous measurements of the system given by the equation

$$\vec{y} = C\vec{x} + \vec{n}_2 \tag{7}$$

where $\vec{y}$ is a k-dimensional output (or measurement) vector, C is a kxn matrix, and $\vec{n}_2$ is a k-dimensional noise vector which is Gaussian distributed with mean 0 and covariance the kxk matrix R. Then, the estimate $\hat{\vec{x}}$ for the state which minimizes the sum of the squares of the estimation errors is given by

$$\begin{aligned} \dot{\hat{\vec{x}}} &= A\hat{\vec{x}} + PC^T R^{-1}(\vec{y} - C\hat{\vec{x}}) && (8) \\ \dot{\hat{P}} &= AP + PA^T - PC^T R^{-1} CP + BQB^T && (9) \end{aligned}$$

where $\hat{P}$ is the estimate of the covariance.

We have dones some preliminary work on setting up STGP to learn the Kalman filter. The terminal and non-terminal sets are

$$\begin{aligned} \mathcal{T} &= \{\mathrm{A, B, C, P\_EST, Q, R, X\_EST, Y}\} && (10) \\ \mathcal{N} &= \{\mathrm{UPDATE\_ESTIMATES, MATRIX\_ADD, MATRIX\_SUBTRACT, MAT\_MAT\_MULT,} \\ &\quad \mathrm{MATRIX\_INVERSE, MATRIX\_TRANSPOSE, MAT\_VEC\_MULT, VECTOR\_ADD,} \\ &\quad \mathrm{VECTOR\_SUBTRACT}\} && (11) \end{aligned}$$

The X-EST and P-EST terminals give the current state and covariance estimates. The Y terminal element contains the measured outputs for the given time step of the current precomputed track. The A, B, C, Q and R terminals are matrices which are constant for a given track. The UPDATE-ESTIMATES function takes two arguments, one of type VECTOR-n and the other of type MATRIX-n-n, and returns type VOID. It works by incrementing X-EST by the time step size times the first argument and incrementing P-EST by the time step size times the second argument. All trees are specified to return type VOID.

Evaluating a parse tree consists of starting with given initial state and covariance estimates and continually executing the tree to update the estimates for each time step. At each time step, the state estimate is compared with the actual state of the track to compute an error estimate for each time step, and the score for a tree is the sum of the squares of the estimation errors.

Currently, our system crashes after evaluating a few hundred parse trees, but we are currently working on fixing this problem and seeing if STGP can learn the optimal solution.

4 Conclusion

Strongly Typed Genetic Programming (STGP) is an extension to genetic programming which fully eliminates the closure constraint necessary for standard genetic programming. It hence allows the user to define functions which take any data types as arguments and return values of any data type. With examples we have shown how STGP produces the solution to problems which would have been virtually impossible for standard genetic programming. While the primary focus of the examples of this paper has been on vector and matrix data types, there are presumably other data types which may also be beneficially used with STGP.

References

[Davis 87] Davis, L. 1987. *Genetic Algorithms and Simulated Annealing.* Pittman.

[Goldberg 87] Goldberg, D.E. 1987. Simple Genetic Algorithms and the Minimal Deceptive Problem, in [Davis 87], pp. 74–88.

[Goldberg 89] Goldberg, D.E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley.

[Kalman 60] Kalman, R.E. 1960. A New Approach to Linear Filtering and Prediction Problems. *Trans. ASME: J. Basic Eng.,* vol. 82, pp. 35–45.

[Koza 92] Koza, J.R. 1992. *Genetic Programming.* The MIT Press.