CONSTRAINTS IN GENETIC PROGRAMMING

Final Report
NASA/ASEE Summer Faculty Fellowship Program – 1995
Johnson Space Center

Prepared By:	Cezary Z. Janikow, Ph.D.
Academic Rank:	Assistant Professor
University & Department:	University of Missouri – St. Louis Department of Mathematics and Computer Science St. Louis, MO 63121
NASA/JSC	
Directorate:	Engineering
Division:	Automation, Robotics and Simulation
Branch:	Intelligent Systems
JSC Colleague:	Dennis Lawler
Date Submitted:	July 20, 1995
Contract Number:	NGT-44-001-800

ABSTRACT

Genetic programming refers to a class of genetic algorithms utilizing generic representation in the form of program trees. For a particular application, one needs to provide the set of functions, whose compositions determine the space of program structures being evolved, and the set of terminals, which determine the space of specific instances of those programs. The algorithm searches the space for the best program for a given problem, applying evolutionary mechanisms borrowed from nature.

Genetic algorithms have shown great capabilities in approximately solving optimization problems which could not be approximated or solved with other methods. Genetic programming extends their capabilities to deal with a broader variety of problems. However, it also extends the size of the search space, which often becomes too large to be effectively searched even by evolutionary methods. Therefore, our objective is to utilize problem constraints, if such can be identified, to restrict this space. In this publication, we propose a generic constraint specification language, powerful enough for a broad class of problem constraints. This language has two elements – one reduces only the number of program instances, the other reduces both the space of program structures as well as their instances. With this language, we define the minimal set of complete constraints, and a set of operators guaranteeing offspring validity from valid parents. We also show that these operators are not less efficient than the standard genetic programming operators if one preprocesses the constraints – the necessary mechanisms are identified.

INTRODUCTION

Evolutionary problem solving simulates nature to search for a solution. First, representation for potential solutions must be defined, along with an evaluation function to quantify different solutions. A number of solutions are simultaneously processed in a simulated population. Individual solutions undergo simulated evolution: Darwinian selective pressure is used for survival determination, mutation is used to alter individual solutions, and crossover is used for information inheritance from usually two parents to one or two offspring.

Genetic programming (GP) [5, 6] is one of the most recent evolutionary methods. It differs from others, such as the well-known genetic algorithms (GAs) [2, 3], by its representation – an individual solution is a high level program, structured as a tree representing the dynamic program structure. This allows, for example, for the learning of analytical functional descriptions [5] – which can not be accomplished with GAs.

A GP application requires specification of the primitive function set and the terminals, which can be used in any combination (the *closure* property) [5]. However, by providing the primitive functions (along with procedural interpretations), one explicitly determines the set of plausible expressions that can be evolved. For example, suppose we need to learn the (unknown) function $y = \sin(x) \cdot \cos(x + 2)$. If we don't know the function, we do not know that the solution will involve trigonometric functions. Therefore, we may decide to use the following function set $\{+, -, *, /\}$. This will prevent the exact solution from being discovered (the *sufficiency* principle states that one assumes that the function set includes all needed functions). The evolved solution would have to be fairly complex for a satisfactory approximation (if possible at all, especially given size constraints imposed on GP programs). On the other hand, if we allow many functions to participate in the primitive set, we explode the search space beyond manageability. Therefore, the GP approach is very sensitive to the user's insights (in addition to being very sensitive to its own parameters). This will hopefully change when methods are developed to constrain the space, to incorporate heuristics, to automatically select/prune the set of primitives, to automatically update the sensitive parameters, *etc.*

An important issue for any problem solving method is that of handling constraints, which are often present. If disregarded, they may lead to infeasible solutions. On the other hand, when properly handled they can reduce the amount of searching required. Constraints can be handled by evolutionary algorithms, especially by genetic algorithms, where most constraint-related research has been done. However, a common problem is that of generality of any approach. Many GA approaches create specialized representations and/or operators, which prohibit invalid solutions from occurring. Examples are a matrix representation for the transportation problem and a permutation sequence for the traveling salesman problem. Even though these approaches are very nicely crafted and are efficient, they are also hand-tailored for the specific problems and must be redone for a new problem, a class of problems, or a specific instance of a problem with different constraints. To avoid that, one needs to provide a more general approach. One such approach has been studied in GAs – penalize solutions which violate the constraints [8]. But this method is not perfect either. Even though it is generic and only requires modification of the evaluation function, it is much less efficient as it still allows explorations of infeasible problem subspaces, wasting resources there. In addition, too relaxed a penalty can still allow generation of infeasible solutions.

Too rigid a penalty may prohibit good solutions from being discovered [7].

The most desirable approach is to provide a constraint-specification language, and then provide built-in mechanisms to handle those constraints, preferably by moving the search into feasible subspaces only. The language should be powerful enough to express a broad range of specific constraints that a particular problem may have. This has been done with GAs for the class of linear constraints over a parameter space [7].

GP offers much greater capabilities than GAs by its variable length parameterized representation. However, as mentioned, it must advance in many directions to enjoy more practical applications. And constraints are an important aspect of such advancement. In this paper, we propose a constraint handling methodology, which is based on the idea of a constraint specification language. This language is presented and enforcement mechanisms are provided. The language is not capable of expressing "any" constraint. However, it is applicable to a broad range of problem constraints. Moreover, we show that the enforcement mechanisms do not reduce the efficiency of the GP algorithm. In fact, the actual search efficiency is greatly improved since the search is now conducted only in the feasible subspace.

The idea is based on restricting both the space of program structures, and on reducing the number of program instances for any particular program structure. In this paper, we overview the various methods for processing constraints in genetic algorithms. Then, we propose a constraint specification language for GP, which is easy to use. Afterwards, we present transformations aimed at reducing the set of constraint specifications to a minimal yet sufficient set, which is easy to enforce. Subsequently, we define mutation and crossover to be "closed" in the program subspace specified by the specifications. That is, given valid parents these operators generate only valid offspring. We also provide a method to initialize the population of programs with only valid instances to ensure that the evolution will be closed in the feasible subspace. Finally, we show that one may implement these operators with the same level of efficiency as the standard operators in unconstrained GP.

CONSTRAINTS IN GP AND GAs

Genetic programming is a special case (or a generalization, depending on the exact definition) of genetic algorithms. Given a problem, a set of constants for the problem (*e.g.*, program variables or actions) is specified, and a set of primitive functions is defined. Eventual program data can be included in the terminal set, or it can be hidden in interpretations of functions. Compositions of functions determine the space of programs which can be expressed this way. What is sought is one of such programs, the one which solves the problem the best way (given some criteria).

Discovery, or rather creation, of the best program is accomplished by evolution. A population of initial programs (possibly random) is set, which then evolve by simulating nature. In this simulation, current programs are evaluated (given problem criteria) and an evaluation-based pressure is used to promote survival of better programs. Programs also undergo mutation, and create offspring by means of crossover.

An obvious problem that GP tries to address is that of searching the infinite space of possible programs. This is done by limiting this space as explained below. Our objective is to further limit this space by additional constraints.

Current State of the Art in Constraint Processing in GP

Except for ADFs (automatic function definitions) and a few tailored applications, there is very little reported work on constraints and methods for handling them in GP [4, 5, 6].

In ADFs, constraints relate to differentiating among different functions and the program being evolved. This has few implications for our objectives. The tailored applications are not more helpful, since they aim at satisfying constraints of very specific functions (usually a single constraint).

In general, the only other constraint used in GP relate to managing the complexity of the evolution. This is done by imposing restrictions on program size. Program size is defined in one of two ways: either by the depth of the tree (proportional to maximum number of nested function calls) or by the total size of the tree (proportional to the total number of function calls). Either case is always handled in one of two different ways:

- abandon the violating program, and keep creating new ones until one that satisfies the constraint is generated (method 1 below),
- use the parent program instead of the violating offspring (method 2 below).

These methods are fine if violations are rare. With the size constraint, this is indeed the case. However, with new constraints, this is unlikely to be true and new methods will have to be investigated.

Various Methods for Handling Constraints

In any evolutionary method (such as GP, but especially in GA), constraints have been, or at least can be, handled in a number of ways. Difficulties come in implementing some of the ways, and in prior determination of which way might be the best for a given problem (or a class of problems). Unfortunately, there is no "silver bullet".

Genetic algorithms enjoy richer methodologies to deal with problem constraints. In addition to those two methods listed for dealing with size constraints in GP, GA researchers have used approaches such as modifications of the interpretation of a particular solution (called *phenotype*), repair algorithms, and penalty functions to penalize invalid solutions. A recently emerging approach is to provide "smart" operators, called *closed*, which are intended to produce only valid offspring from valid parents. Such operators have been proposed for GA parameter optimization with linear constraints [7].

Objectives and Methods

Because possible constraints are endless, it is impossible to provide constraint handling methods for specific constraints ahead of time. On the other hand, we want to avoid having to provide such methods separately for each problem, whenever a constraint appears. Therefore, our objective is to design a constraint specification language able to express constraints encountered in a broad range of applications. Then, we could have generic mechanisms processing those constraints so that

- only valid programs are evolved,
- the space of valid programs is constrained to reduce the effective search space.

EXPLORATION OF CONSTRAINT SPECIFICATIONS

Search Space in GP

GP's problem space is defined by terminals T and functions F, where each $f_i \in F$ has a fixed arity a_i. Function compositions determine the space of possible program structures. Because the space is infinite, it is normally restricted by the aforementioned two methods – restrict the number of nodes or restrict the maximal depth of a program. Terminals are values which can appear in terminal nodes. They have no implications for program structures. Instead, given a specific program structure, which serves as a template, terminals determine the space of possible program instances. This space, for a particular program structure, can again be infinite if infinite sets are allowed (such as real numbers). Therefore, the space of all programs is determined by both F and T, and this space can be infinite in two dimensions. One of these possibly infinite dimensions , the space of program structures, is normally restricted by the size constraints. The other dimension, program instances of a program structure, is restricted by the finite accuracy of computer representations. It is assumed that this space contains the sought program. This assumption is the basis for GP problem solving – the *sufficiency* assumption.

T- and F- Constraint Specifications

We use two different kinds of constraint specifications, but, as we will see, they are not completely independent. These are syntactic *T-specifications* and semantic *F-specifications*. Moreover, we will show that not all of them are really needed – after certain transformations, only a few are sufficient to express exactly the same constraints. However, it will generally be much easier to express problem constraints with the full specification language, while it will be much easier to devise mechanisms for the minimal set.

T-specifications are based on domains for function arguments and on function ranges.

Definition 1 *Let us define $\Rightarrow$ to stand for domain compatibility. That is, $X \Rightarrow Y$ means that X can replace Y, where both X and Y stand for sets of values (finite or infinite) allowed for domains or returned as function ranges.*

Definition 2 *Define the following T-specifications (syntactic constraints):*

1. T^{Root} *– the set of values allowed at the Root, or the set of values allowed to be returned by the evolved program, that is by functions appearing at the Root.* T^{Root} *actually specifies both a domain (for the root node) and a range (for the program).*
2. $T_* - T_i$ *is the range of* f_i*, that is the set of values returned by the function* f_i*.*
3. $T_*^* - T_i^j$ *is the domain for the* j^{th} *argument of* f_i*, that is the set of values allowed there (which may be returned by functions used as this argument).*
4. $T_* \stackrel{?}{\Rightarrow} T_*^*$ *– compatibilities between ranges and domains*
5. $T_* \stackrel{?}{\Rightarrow} T^{Root}$ *– compatibilities between function ranges and the program range*

where '?' indicates that the compatibility can be straight or it can be negated.

T-specifications reduce both the space of program structures and the space of instances of those structures. Therefore, they are very powerful constraints. GP uses only one terminal set – any value from that set may appear in any leaf of a program tree. Obviously, this is not true in actual programs – different functions take different arguments. *T-specifications* allow expressing such differences, thus allowing reduction in the space of program instances per program structure. Moreover, some *T-specifications* also implicitly restrict what function can call other functions, effectively reducing the space of possible program structures. For example, if the domain for a function argument is fixed, then the value for that argument may not be obtained from a function with an incompatible range. Therefore, some *T-specifications* can be seen analogous to *function prototypes* available in high level languages.

However, syntactic fit does not necessarily mean that a function *should* call another function. One needs additional specifications based on program *semantics*. These are provided by means of *F-specifications*, which further restrict the space of program structures (but not program instances).

Definition 3 *Define the following F-specifications (semantic constraints):*

6. F^{Root} *– the set of functions disallowed at the Root.*
7. F_* *—* F_i *is the set of functions disallowed as direct callers to* f_i *(generally, a function is unaware of the caller; however, GP constructs a program tree, which represents the dynamic structure of the program).*
8. $F_*^* - F_i^j$ *is the set of functions disallowed as* arg_j *to* f_i.

Example 1 *Assume a function* $(if\ arg_1\ arg_2\ arg_3)$, *interpreted as: if* arg_1 *evaluates to true, return the evaluation of* arg_2, *else return the evaluation of* arg_3 *(assume both of which evaluate to real numbers). One needs to specify that* arg_1 *could only be terminals which are boolean values, or only functions which return boolean values. Assuming that* $T = R^1 \cup \{T, F\}$, *one may specify* $T_{if}^1 = \{T, F\}$. *Because* R^1 *is not compatible with* $\{T, F\}$, *only elements of the latter can be placed there.*

Proposition 1 $X \Rightarrow Y \leftrightarrow X \subseteq Y$.
:: $X \Rightarrow Y$ *means that in places where values from* Y *are valid one may place any value from* X, *or any function returning a value from* X. *To guarantee that no out-of-domain values are used for the original* Y, X *may not contain values not found in* Y. *Therefore, it must be a subset of* Y, *or it must equal* Y.

Using known properties of $\subseteq$, domain compatibilities could be automatically computed (giving compatibility *T-specifications* #4 & #5), as long as these are restricted to syntactic constraints.

Example 2 *Assume two sets:* $T_1 = \{1, 2, 3\}$ *representing masses of physical objects in kilograms, and* $T_2 = \{1, 2\}$ *representing times in seconds. Thus one may conclude that* $T_2 \Rightarrow T_1$ *since* $\{1, 2\} \subset \{1, 2, 3\}$. *But by observing the interpretations of these objects, an obvious conclusion is that* $T_2 \not\Rightarrow T_1$, *but this is based on interpretation of these sets, which is left to F-specifications.*

Rules on T- and *F-specifications*

Given the above *T-specifications* and *F-specifications*, which can be used to express problem constraints, an obvious issue is that of possible redundancies, or that of existence of sufficiently minimal specifications. We answer these questions in this section. Surprisingly, after certain static transformations, only a subset of *T-* and *F-specifications* will turn out to be sufficient to express all *T-* and *F-specification* constraints. This observation is extremely important, as it will allow efficient constraint enforcement mechanisms after initial preprocessing. The sufficient minimal set is thus important for efficient constraint processing, but not for constraint specifications – specifications are more easily expressed with the original *T-specifications* and *F-specifications*. This is why we need both, along with the necessary transformations.

The first step is is to extend *F-specifications*.

Definition 4 *Define 'complete' T-specifications as those that list all elements of Definition 2, including ranges and domains for all functions and their arguments and compatibilities between all pairs range–domain and range–program range.*

Proposition 2 *The following F-specification constraints are implied by complete T-specifications:*

$$\forall_{f_k \in F}(T_k \not\Rightarrow T_i^j \rightarrow f_k \in F_i^j)$$

$$\forall_{f_k \in F}(T_k \not\Rightarrow T^{Root} \rightarrow f_k \in F^{Root})$$

:: If f_k returns a range which is not compatible with the domain for a specific function argument, then f_k cannot be used to provide values for the argument. The same applies to values returned from the program.

Proposition 2 is very important because the compatibility *T-specifications* (#4 & #5) can be automatically generated from other *T-specifications*, and according to the rule, they can be automatically translated to *F-specifications*. The latter, as we will see, are easier to handle.

Note that the opposite of these implications is not true since some *F-specifications* are based solely on interpretations. In other words, it is not true that $f_k \in F_i^j \rightarrow T_k \not\Rightarrow T_i^j$. Note that the following is not true either: $\forall_{f_k \in F}(T_k \Rightarrow T_i^j \rightarrow f_k \notin F_i^j)$ (see Example 2). Fortunately, the first implication is sufficient for us as it tells us that properly extended F_*^* and F^{Root} specifications subsume the $T_* \not\Rightarrow T_*^*$ and $T_* \not\Rightarrow T^{Root}$ *T-specifications*.

Example 3 *Suppose $f = \{f_1, f_2\}$, $a_1 = 1, a_2 = 1$. Also suppose f_1 returns real-valued numbers ($T_1 = R^1$), and f_2 takes boolean arguments ($T_2^1 = \{F, T\}$). Because $T_1 \not\Rightarrow T_2$, we may conclude that f_1 cannot be placed as the argument to f_2: $f_1 \in F_2^1$.*

Definition 5 *If F-specifications explicitly satisfy Proposition 2 then call them 'T-extensive' F-specifications. If F-specifications do not explicitly satisfy Proposition 2 for any function $f_k \in F$, then call them 'T-intensive' F-specifications.*

In other words, *T-intensive F-specifications* list only some additional constraints – which cannot be derived from *T-specifications*. *T-extensive F-specifications*, on the other hand, are those semantics-based constraints extended by syntactic constraints on function calls.

For now, we will look at redundancies among *F-specifications*.

Proposition 3 *Suppose $f_k \in F$ and F-specifications are T-extensive. Then*

$$\forall_{f_i \in F}(f_k \in F_i \leftrightarrow \forall_{j \in [1,a_k]} f_i \in F_k^j)$$

:: If a function f_i cannot call f_k, then f_k will never be called by f_i. Also, if f_k is never called from f_i, it must not be called from any of f_i's arguments.

With Proposition 3 one may wonder whether we need both F_*^* and F_* constraints – they seem equivalent. The next rule says they are not.

Proposition 4 *Suppose $f_k \in F$ and F-specifications are T-extensive. Then*

$$\forall_{f_i \in F}(\exists_{j \in [1,a_i]} f_k \in F_i^j \not\Rightarrow f_i \in F_k)$$

The implication is true only when Proposition 3 applies.
:: If f_k cannot be called from f_i by its j^{th} argument, it may possibly be allowed as another argument (unless, according to Proposition 3, it cannot be called from any of the arguments).

Even though they are not equivalent, both are not needed either. It turns out that F_*^* *F-specifications* are stronger.

Definition 6 *If F-specifications explicitly satisfy Proposition 3, call them 'F-extensive' F-specifications. If F-specifications do not include any F_* constraints, call them 'F-intensive' F-specifications.*

Proposition 5 *F-intensive F-specifications are sufficient to express all possible F-specifications.*
:: According to Proposition 3, $f_k \in F_i$ can be deduced when f_i is excluded from all arguments of f_k. According to Proposition 4, it can happen only when Proposition 3 applies. Therefore, F-intensive F-specifications provide sufficient information to produce F-extensive F-specifications.

We now return to the question of *T-specifications* vs. *F-specifications*. We have seen that *T-intensive F-specifications* provide restrictions on function calls based on interpretations, and that they can be extended to *T-extensive F-specifications*, which also take syntax into account. One question that comes to mind is: do we still need *T-specifications* after they have been used to produce *T-extensive F-specifications*? In other words, is there any constraint in *T-specifications* which is not expressed with *T-extensive F-specifications*? The answer is 'no' for certain *T-specifications*.

Proposition 6 *T-extensive F-specifications are sufficient to express constraints imposed by compatibility (#4 & #5) and T_* (#2) T-specifications.*
:: Let us look at compatibilities of the form $T_k \stackrel{?}{\Rightarrow} T_i^j$. Proposition 2 says that the negated forms ($\not\Rightarrow$) are all expressed in T-extensive F-specifications. However, the straight form ($\Rightarrow$) can be superseded by F-specifications, which provide additional constraints based on interpretations. Thus, if $f_k \in F_i^j$, then the corresponding T-specification is irrelevant. On the other hand, if $f_k \notin F_i^j$ (in the T-intensive form), then we have two cases:

- *if $T_k \not\Rightarrow T_i^j$, then according to Proposition 2 we put f_k into F_i^j: $f_k \in F_i^j$ in T-extensive forms*
- *if $T_k \Rightarrow T_i^j$, then we have no reason to extend F-specifications– thus, $f_k \notin F_i^j$*

The same can be argued for $T_k \stackrel{?}{\Rightarrow} T^{Root}$. As to T_* T-specifications , they are sets of values returned by functions. Therefore, they place restrictions on function calls. But, F-extensive F-specifications express all possible restrictions on function calls. Said differently, T_* is only used for other specifications.

Definition 7 *Define T-extensive F-intensive F-specifications as the 'normal' form.*

Theorem 1 *(Fundamentals of T– and F-specification constraints) Even if the user provides only T-F-intensive F-specifications, T-F-extensive F-specifications can be computed, and along with domains and the program range they are sufficient to express all T– and F-specification constraints. Moreover, just the normal F-specifications along with domains and the program range are sufficient as well.*
:: It follows from Propositions 5 and 6.

Based on Theorem 1, we may now restrict our discussion to *F-specifications* only, assuming that these are in the *normal* form. To make sure they are, a simple preprocessing mechanism suffices.

EXPLORATION OF CONSTRAINT HANDLING METHODS

We propose to implement the specified constraints into "smart" operators. To do so, we must define operators "closed" in the valid program structure – from valid parents always generate valid offspring. This will also require an initialization procedure with valid programs.

Definition 8 *In the program tree, we call 'function nodes' all nodes which correspond to a function. In this case, we say that the function labels the node. All other nodes are called 'terminal nodes'.*

Definition 9 *Define $\mathcal{T}_N$ to be the set of values which can replace node N. That is, $\mathcal{T}_N$ is the set of values that the node can assume without invalidating, w/respect to T-specifications and F-specifications , the program tree containing that node.*

Definition 10 *Define $\mathcal{F}_N$ to be the set of functions which can replace node N. That is, $\mathcal{F}_N$ is the set of functions which can label that node without invalidating, w/respect to T-specifications and F-specifications , the program tree containing that node.*

For terminal nodes, we cannot determine what other possible values can it contain by just looking at the node. We must look at the parent of the node (unless it is the *Root*). For function nodes, we could either use the set of values returned by the function labeling that node (T_i for f_i). However, after replacing the function node with a terminal node, we

would anyway have to look at the context where the node appears. Therefore, we decided to use the context information even for function nodes.

As the subsequent rules state, the above sets not only can be efficiently computed, but some can also be guaranteed to be non-empty under certain conditions, which hold for GP. Moreover, in the next section we will see that these sets can be precomputed for all possible node types, and that functions to extract random elements of these sets can be precomputed as well. This will lead to a very efficient enforcement of these constraints.

Proposition 7 *Assume a node N is the j^{th} argument of f_i and F-specifications are normal. Then,*

$$\mathcal{T}_N = T_i^j$$

$$\mathcal{F}_N = \{f_k | (f_k \in F) \wedge (f_k \notin F_i^j)\}$$

:: Any value that does not invalidate the domain T_i^j is OK. Any function that is not explicitly excluded from F_i^j is OK. This is so because if $f_i \in F_k$, that is if f_i cannot be accepted as a caller to f_k, then according to Proposition 3 $f_k \in F_i^j$, but it is not.

Proposition 8 *Assume a node N is the Root and F-specifications are normal. Then,*

$$\mathcal{T}_N = T^{Root}$$

$$\mathcal{F}_N = \{f_k | (f_k \in F) \wedge (f_k \notin F^{Root})\}$$

:: Arguments are analogous to those for Proposition 7, except that the Root provides the constraints.

Proposition 9 *$\mathcal{T}_N \neq \emptyset$ for any terminal node in any valid program.*
:: The valid program does not change when the terminal node is replaced with itself.

Proposition 10 *As long as any function returns a value (as it is in GP), $\mathcal{T}_N \neq \emptyset$ for any function node in any valid program.*
:: If the function node is labeled with f_i, then it can be replaced with any terminal from T_i. This set is not empty as long as each function returns at least one value.

Proposition 11 *For any function node N of any valid program, $\mathcal{F}_N \neq \emptyset$.*
:: If the node is labeled with f_i, then $f_i \in \mathcal{F}_N$.

Note that $\mathcal{F}_N$ is not guaranteed to be non-empty for terminal nodes. That is, some terminals may only be used for computations, but will never be computed.

Example 4 *Suppose $F = \{f_{[]}\}$, and $f_{[]}$ returns the closest integer to its real-valued argument. Then, $T_{[]} = I$, $T_{[]}^1 = R^1$, and $T_{Root} = I$. Also, $(f_{[]} \in F_{[]}^1) \wedge (f_{[]} \in F_{[]})$ (in the T-F-extensive form), but only $(f_{[]} \in F_{[]}^1)$ is sufficient (in the normal form).*

For the program ($f_{[]}$ 3.27), the terminal node 3.27 has $\mathcal{T} = I$ and $\mathcal{F} = \emptyset$.

We can now define closed operators. We assume that all random numbers are taken from a uniform distribution. For any node N, denote

- $r_N^{\mathcal{T}}$ to be a random element from $\mathcal{T}_N$
- $r_N^{\mathcal{F}}$ to be a random element from $\mathcal{F}_N$ (assuming that it is non-empty)

For any terminal node N, denote

- v_N to be the current value from that node

Mutation

Assume that node N is chosen for mutation. This selection can be based on a fixed probability of mutating any *allele* in all chromosomes (often called *post mutation* in GP), or on selecting a random *allele* in a given selected parent (*normal mutation* in GP).

Operator 1 *(mutation) If a node N is selected for mutation, then replace it with $r_N^{\mathcal{T}}$ with probability p_m^1, or with $r_N^{\mathcal{F}}$ with $1 - p_m^1$. If $r_N^{\mathcal{F}}$ is used, then recursively repeat exactly the same Operator 1 on all arguments of the selected function.*
If $r_N^{\mathcal{F}}$ is needed for the node N and the $\mathcal{F}_N$ set is empty, try another random node from the same parent (in normal mutation) or abandon the operation (in post mutation). If $r_N^{\mathcal{F}}$ is needed for any descendent of N and the $\mathcal{F}_N$ set is empty, use $r_N^{\mathcal{T}}$ instead.

Proposition 12 *For any valid parent program, mutation Operator 1 is guaranteed to take place as long as $p_m^1 > 0$. For a function node, Operator 1 is guaranteed to take place immediately. Moreover, all T- and F-specification constraints are guaranteed to be preserved.*
:: The parent is valid. According to Propositions 9 and 10, the set $\mathcal{T}^N$ is never empty. Therefore, as long as this set is allowed in mutation ($p_m^1 > 0$), mutation will eventually take place on any node. However, if N is a function node, then according to Propositions 10 and 11 both $\mathcal{T}_N$ and $\mathcal{F}_N$ are non-empty, so mutation will immediately take place regardless of p_m^1. The mutation sets are computed based on normal F-specifications, which are sufficient according to Theorem 1.

Crossover

Because crossover with two offspring can be accomplished with two crossover operations, each with one offspring, we will define crossover with one offspring. In unconstrained GP, there are no specific constraints. Therefore, crossover is reduced to finding two random crossover points. In constrained "smart" crossover, the choices of plausible crossover points can be highly reduced. Requiring that two offspring can be generated from the same two crossover points further reduces chances of finding such points, but can be done if necessary.

Definition 11 *Define $\mathcal{S}_{N,x}$ to be the set of nodes from $parent_x$ which can replace a given node N selected for crossover.*

Proposition 13 *For crossover at node N_1 in $parent_1$, and another $parent_2$,*

$$\mathcal{S}_{N_1,2} = \left\{ N_2 \;\middle|\; \begin{array}{ll} v_{N_2} \Rightarrow \mathcal{T}_{N_1} & \text{if } N_2 \text{ is a terminal node in } parent_2 \\ f_i \in \mathcal{F}_{N_1} & \text{if } N_2 \text{ is a function node labeled } f_i \text{ in } parent_2 \end{array} \right\}$$

:: Crossover may be seen analogous to mutation – from $parent_2$ select those crossover nodes which are also allowed in mutation for N_1. And $\mathcal{T}_N$ and $\mathcal{F}_N$ are terminals and functions that the node N_1 can mutate into. If $N_1 = Root$ then $\mathcal{T}_{N_1}$ and $\mathcal{F}_{N_1}$ are terminals and functions that the node $Root_1$ can mutate into.

Operator 2 *(crossover) If $parent_1$ and $parent_2$ are two crossover parents, select a random crossover point N_1 in $parent_1$, except that internal nodes have collective probability of p_c^1 and leaves have collective probability $1 - p_c^1$ (following standard GP practice of directing crossover into internal nodes). Based on whether N_1 is the Root, apply Proposition 13 to compute $S_{N_1,2}$. If the set is not empty, then select a random node N_2 (leaves and internal nodes may be given distinct probabilities with p_c^1), and replace the subtree starting with N_1 with that staring with N_2. If the set is empty, try another crossover point N_1 from $parent_1$.*

Proposition 14 *For any two valid parents, Operator 2 is guaranteed to find valid crossover, and the operation will satisfy $T-$ and F-specifications.*
:: Both parents are valid. Therefore, replacing them wholly will produce a valid offspring. Moreover, the offspring is created by replacing a subtree with another subtree from a set computed according to $T-$ and F-specifications. Therefore, any offspring will satisfy these constraints.

Feasible Initialization Procedure

Operator 3 *(initialization) Initialize the population by growing chromosomes starting each with a random terminal node N such that $v_N \in \mathcal{T}_{Root}$, and then applying Operator 1 to that node.*

Proposition 15 *The above initialization routine Operator 3 will only generate individuals which are valid with respect $T-$ and F-specification constraints.*
:: Operator 1 guarantees a valid offspring from a valid parent (Propositions 7, 8). And the initial terminal node is valid as the Root.

IMPLEMENTATION

Constraint preprocessing

We do not need terminals T to be explicitly given as in GP – T_*, T_*^*, and T^{Root} will determine individual sets. The preprocessing needed to ensure that *F-specifications* are *normal* and that our operators can apply can be described as follows:

1. Read T^{Root}, T_* ranges for functions of F and T_*^* domains for their arguments
2. Read *T-specification* compatibility constraints of the form $T_* \stackrel{?}{\Rightarrow} T_*^*$ and $T_* \stackrel{?}{\Rightarrow} T^{Root}$ (not necessary if computed automatically)
3. Read (at least *T-intensive F-intensive*) *F-specifications*

4. Compute *normal F-specifications*

5. Produce functions for $r^{\mathcal{T}}$ and $r^{\mathcal{F}}$ for all necessary sets.

Given this preprocessing mechanism, the defined operators can be used in any GP.

Implementation

Proposition 16 *$r^{\mathcal{T}}$ and $r^{\mathcal{F}}$ can be precomputed, as part of the preprocessing mechanism, into functions returning random elements of those sets.*
:: For mutation, we directly need $r^{\mathcal{T}}_N$ and $r^{\mathcal{F}}_N$. For any mutation, it is determined in one step which of the two is needed. Based on whether N is the Root or not, Proposition 8 or 7 gives us exactly the sets from which the random element is selected. There is a fixed number of these sets: there are exactly $1 + \prod_{f_i \in F}(a_i)$ of each $\mathcal{T}$ and $\mathcal{F}$ sets. All $\mathcal{F}$ sets are always finite with up to $|F|$ elements, and $\mathcal{T}$ is either finite, or infinite when domains such as reals are used. Moreover, these sets never change as GP operates. For the $\mathcal{F}$ sets, the elements can be enumerated and $r^{\mathcal{F}}$ can be compiled into a function returning a random function from each enumerated set. For the $\mathcal{T}$ sets which are infinite, $r^{\mathcal{T}}$ can be precompiled to returning a random entry from the domain. For finite sets, the elements can be enumerated again and $r^{\mathcal{T}}$ can be compiled into a function returning a random element from each enumerated set. For sets which are unions of finite or infinite subsets, one may first determine which class of subsets to choose from (assuming that we provide some measures comparing cardinalities of finite and infinite sets, or the user provides such information), and then apply one of the two above techniques.

For crossover, we need to use the $\mathcal{S}_{N,a}$ sets. However, at each time we know whether the node N_a is a terminal or a function node, at which moment the problem reduces to the same as in mutation – selecting random entries from the appropriate $\mathcal{T}$ or $\mathcal{F}$ set. Moreover, if P^1_c is used, the elements may be divided into two groups from which to select the random entry – p^1_c would determine which group to use.

Theorem 2 *(Implementation theorem for GP) The defined mutation and crossover operations not observing size constraints are as efficient as the standard operators in GP, when implemented with the preprocessing mechanism.*
:: In GP, mutation generates a random function from F or a random element of T. Crossover selects a random subtree. It follows directly from Proposition 16 that in our approach any mutation or crossover can be accomplished by selecting a random entry from a fixed set, even though the sets are more plentiful. However, for any node it is deterministic, in a fixed time, which set should be used.

Conjecture 1 *Provided T-specifications and F-specifications are the maximal constraints that can be implemented into a generic constrain processing methodology in GP without invalidating Theorem 2.*
:: Other constraints will require information about a node position in a tree – processing complexity would be a function of tree depth.

References

[1] Lawrence Davis (ed.). *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, 1991.

[2] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison Wesley, 1989.

[3] Holland, J. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.

[4] Kenneth E. Kinnear, Jr. (ed.) *Advances in Genetic Programming.* The MIT Press, 1994.

[5] John R. Koza. *Genetic Programming.* The MIT Press, 1992.

[6] John R. Koza. *Genetic Programming II.* The MIT Press, 1994.

[7] Zbigniew Michalewicz & Cezary Z. Janikow. *"GENOCOP: A Genetic Algorithm for Numerical Optimization Problems with Linear Constraints".* To appear in *Communications of the ACM.*

[8] J.T. Richardson, M.R. Palmer, G. Liepins & M. Hilliard. *"Some Guidelines for Genetic Algorithms with Penalty Functions".* In *Proceedings of the Third International Conference on Genetic Algorithms.* Morgan Kaufmann, 1989.