A Methodology for Processing Problem Constraints in Genetic Programming

Cezary Z. Janikow*
Department of Mathematics and Computer Science
University of Missouri - St. Louis
emailto:janikow@radom.umsl.edu

November 6, 1997

Abstract

Search mechanisms of artificial intelligence combine two elements: representation, which determines the search space, and a search mechanism, which actually explores the space. Unfortunately, many searches may explore redundant and/or invalid solutions. Genetic programming refers to a class of evolutionary algorithms based on genetic algorithms but utilizing a parameterized representation in the form of trees. These algorithms perform searches based on simulation of nature. They face the same problems of redundant/invalid subspaces. These problems have just recently been addressed in a systematic manner. This paper presents a methodology devised for the public domain genetic programming tool *lil-gp*. This methodology uses data typing and semantic information to constrain the representation space so that only valid, and possibly unique, solutions will be explored. The user enters problem-specific constraints, which are transformed into a normal set. This set is checked for feasibility, and subsequently it is used to limit the space being explored. The constraints can determine valid, possibly unique space. Moreover, they can also be used to exclude subspaces the user considers uninteresting, using some problem-specific knowledge. A simple example is followed thoroughly to illustrate the constraint language, transformations, and the normal set. Experiments with boolean 11-multiplexer illustrate practical applications of the method to limit redundant space exploration by utilizing problem-specific knowledge.

*Supported by a grant from NASA/JSC: NAG 9-847.

1 Preliminaries

Solving a problem of the computer involves two elements: representation of the problem, or that for its potential solutions, and a search mechanism to explore the space spanned by the representation. In the simplest case of computer programs, the two elements are not explicitly separated and instead are hard-coded in the programs. However, separating them has numerous advantages such as reusability for other problems which may require only modified representation. This idea has been long realized and practiced in artificial intelligence. There, one class of algorithms borrows ideas from nature, namely population dynamics, selective pressure, and information inheritance by offspring, to organize its search. This is the class of *evolutionary algorithms*.

Genetic algorithms (GAs) [2, 3, 4] are the most extensively studied and applied evolutionary algorithms. A GA uses a population of chromosomes coding individual potential solutions. These chromosomes undergo a simulated evolution facing Darwinian selective pressure. Chromosomes which are better with respect to a simulated environment have increasing survival chances. In this case, the measure of fit to this environment is based on the quality of a chromosome as a solution to the problem being solved. Chromosomes interact with each other via *crossover* to produce new offspring solutions, and they are subjected to *mutation*.

Most genetic algorithms operate on fixed-length chromosomes, which may not be suitable for some problems. To deal with that, some genetic algorithms adapted variable-length representation, as in machine learning [3, 5]. Moreover, traditional genetic algorithms use low-level binary representation, but many recent applications use other abstracted alphabets [2, 5].

Genetic programming (GP) [7, 8, 9] uses trees to represent chromosomes. At first used to generate LISP computer programs, GP is also being used to solve problems where solutions have arbitrary interpretations [7]. Tree representation is richer than that of linear fixed-length strings. However, there is a price to pay for this richness.

In general, the number of trees should equal the number of potential solutions, with one-to-one mapping between them. Unfortunately, this is hardly ever possible. Because we need a mapping for each potential solution, the number of trees will tend to be much larger, with some of them being redundant or simply invalid. Therefore, some means of dealing with such cases, such as possibly avoiding the exploration of such extraneous trees, are desired. While for some problems some *ad-hoc* mechanisms have been proposed [7, 8], there is no general methodology. Our objective is to provide a systematic means, while making sure that the means do not increase the overall computational complexity. In this paper, we present a method suitable for, and implemented with, a standard GP tool *lil-gp*. This methodology is a somehow weaker version of [6], modified specifically for *lil-gp*.

lil-gp is a tool [11] for developing GP applications. Its implementation is based on the standard GP *closure* property [8], which states that every function can call any other function and that any terminal can provide values for any function argument. Even though this is usually called a "property", it is in fact a necessity in the absence of other means for dealing with invalid trees.

Almost any application imposes some problem-specific constraints on the placement of elements in solution trees. Invalid solutions can be re-interpreted as redundant solutions (as done to force *closure*), or they can be assigned low evaluations, practically causing their extinction (*penalty* approach). Both approaches may face potential problems. Too many *ad-hoc* redundancies may easily change problem characteristics (problem *landscape*). Too many extinction-bound solutions waste computational resources and may cause premature convergence (*over-selection* in GP [8]). Recently, other methods have been explored and proposed. For example, Montana has developed means for ensuring that only valid trees evolve (*Strongly Typed Genetic Programming* - STGP [10]), and we independently proposed a similar methodology for processing more arbitrary constraints in *Constrained Genetic Programming* (CGP) [6]. CGP, in addition to providing means for avoiding exploration of invalid subspaces, also provides for specification/avoidance of both redundant subspaces as well as subspaces which are perfectly valid but some problem-specific heuristics suggest to exclude them from being desired solutions.

The objectives of CGP is to provide means to specify syntax and semantic constraints, and to provide efficient systematic mechanisms to enforce them [6]. We have just implemented a pilot tool[1], which incorporates CGP with the widely used GP tool *lil-gp1.02* (*lil-gp* allows forrest chromosomes, which for computer programs corresponds to program modules - our current methodology deals with a single tree, but it is currently being extended). This paper describes CGP applied to *lil-gp* (called *CGP lil-gp*).

Even though this paper is not intended to compare STGP with CGP, it is worthwhile to point out that CGP ensures that the extra processing does not change the overall complexity of the basic GP mechanisms. Moreover, CGP allows more user-friendly front-end for constraint specifications, with a transformation aimed at reducing the constraints to a minimal set. It also allows various constraints, such as syntax- and semantics-based. Finally, CGP's crossover is more powerful since it allows more feasible offspring from the same two parents. A more systematic comparison will be presented separately.

In section 2, we overview the problems CGP attempts to alleviate. In section 3 we present the CGP methodology for *lil-gp*, along with a complexity analysis. A simple example is used to illustrate the processing from constraint specification to generation of the minimal set. This section can be omitted by readers not interested in technical details. In section 4 we present initial experiments designed to illustrate *CGP lil-gp*'s application to deal with redundant/undesired search spaces using the 11-multiplexer problem. This experiment is only intended to illustrate how problem-specific knowledge can be expressed with the constraint language, and what the implications of restricting the search are. However, some important observations are made in section 4.10.

[1]fttp://radom.umsl.edu

2 State-Space Searches and GP Search Space

In artificial intelligence, solving a problem on the computer involves searching the collection of possible solutions. For example, solving a two-dimensional integer optimization problem with both domains [1,100] would involve searching through the space of 10,000 solutions (one for each pair). This search may be random, enumerated exhaustive, or heuristic [1]. However, in most practical problems of interest to artificial intelligence, the space of potential solutions is too large to be explicitly retained and effectively randomly or exhaustively searched by an algorithms. Instead, the space is defined by implicit means, often by transition operators generating new states from existing ones, along with a set of currently explored solutions. Given a complete set of operators, some control strategy is then used to manage the search. Such approaches are called *state-space searches* in artificial intelligence [1].

Evolutionary algorithms utilize state-space searches. The subspace being explored is retained in the population of chromosomes. Genetic operators, such as mutation and crossover, generate new solutions from the existing ones. The control is stochastic, promoting exploration of "better" subspaces (additional heuristics may be used to further guide the search, as in [5]).

In GP, a set of functions and a set of terminals are defined. Elements of these label the internal and the external nodes, respectively. Interpretations of those elements are given by providing implementations for evaluating nodes labeling them. Then, a generic interpreter uses those interpretations to evaluate a tree by following a standard traversal. For example, a root node having three subtrees must be labeled with a function having three arguments. These subtrees are evaluated recursively, and their values are used as arguments to the function labeling the root.

Following evolutionary algorithms, GP generates a population of random trees, using the primitive elements. Then, all trees are evaluated in the environment (using functional interpretations with the interpreter, and the problem at hand). Crossover and mutation are used to generate new trees in the population, from parents selected following Darwinian principles. GP also allows another operator, *selection*, which simply copies chromosomes to the new population.

Since all trees are evolved from the primitive elements, these must be sufficient to generate the sought tree. The assumption that this is indeed the case is called the *sufficiency* principle [8]. However, in general to satisfy sufficiency a large number of functions must be given. This unfortunately exponentially explodes the search space. In up-to-date applications, this is dealt with by providing "the right" functions and terminals. Obviously, in many cases this may not happen, and the search space will explode nevertheless. To deal with this potential problem, as well as its current weak manifestation, practical size restrictions are imposed on the trees. Unfortunately, the more rigid the restriction, the more likely that some important solutions may be excluded.

In the next section we will present a methodology to utilize constraints to prune implicitly identified subspaces. In general, the constraints we propose for the pruning include both syntactic and semantic elements. Syntactic constraints include typing function arguments, values returned by functions, and individual terminals. These are similar to those of Montana [10]. Semantic constraints are additional restrictions based on function or terminal interpretation. The methodology presented here is a weaker version of that presented in [6], but it is the one that has been implemented with *lil-gp*.

3 CGP Methodology

3.1 Constraint specifications

In *lil-gp* [11], functions and terminals fall into three categories. Let us call them functions of type I, II, and III (as described below), and sets of those functions will be denoted as F_I, F_{II}, and F_{III}. Unless explicitly stated, all references to functions imply all function types (denoted F), and all references to terminals imply functions of type II and III (denoted T). Borrowing [11]'s terminology, we have:

I. *Ordinary functions.* These are functions of at least one argument, thus they can label internal nodes, with the number of subtrees corresponding to the number of arguments.

II. *Ordinary terminals.* These are functions of no arguments. Therefore, they can label external nodes. However, they are not instantiated in trees but rather during interpretation. In other words, these terminal values are provided by the environment (as for a function reading the current temperature).

III. *Ephemeral random constant terminals.* These are functions of no arguments, which are instantiated individually in each tree, thus the values are independent of the environment.

In *lil-gp* [11], terminal sets for type III are not *extensively* defined. Instead, they are defined by generating functions, which return uniform random elements from the appropriate ranges. Ranges for functions of type I and type II and not explicitly defined either.

In [6] we defined the notion of domain/range compatibility (denoted here $\Rightarrow$), which can be used to infer validity of using functions and terminals as arguments to other functions. That notion, based on sets, allows automated processing of such compatibilities. With *lil-gp*, these capabilities cannot be automated since no explicit sets are used, and the resulting methodology is somehow weaker (section 4 gives an example). Therefore, all compatibility specifications are left to user's responsibility (similarly to Montana's approach [10]). This unfortunately means that the user must be trained in the domain. Fortunately, our preprocessing offers a user-friendly method to specify constraints which method can deal with inconsistent and/or redundant specifications.

Definition 1 *Define the following* Tspecs *(syntactic constraints):*

1. T^{Root} – *the set of functions which return data type compatible with the problem specification.*

2. $T_*^* - T_i^j$ *is the set of functions compatible with the jth argument of* f_i.

In terms of a labeled tree, T^{Root} is the set of functions which, according to data types, can possibly label the *Root* node. T_i^j is the set of functions that can possibly label the j^{th} child node of a node labeled with f_i.

Following *closure*, *lil-gp* allows any function of type I to label any internal node, and any function of type II and III to label any external node. Obviously, in general, different functions take different arguments and return different ranges. *Tspecs* allow expressing such differences, thus allowing reduction in the space of tree structures and tree instances. Moreover, some *Tspecs* also implicitly restrict what function can call other functions. *Tspecs* are analogous to *function prototypes* and *data typing* in the context of tree-like computer programs, and thus they are similar to Montana's type restrictions [10]).

Example 1 *Assume* $F_I = \{f_1, f_2, f_3\}$ *with arities 3, 2, and 1, respectively. Assume* $F_{II} = \{f_4\}$ *and* $F_{III} = \{f_5, f_6, f_7\}$. *Assume that the three type III functions generate random boolean, integer, and real, respectively. Assume* f_4 *reads an integer. Assume* f_1 *takes boolean and two integers, respectively, and returns a real. Assume* f_2 *takes two reals and returns a real. Assume* f_3 *takes a real and returns an integer. Also assume that the problem specifications state that a solution program should compute a real number (what the problem might be is irrelevant here). The example assumes that integers are compatible with reals, while booleans are not compatible with either (different than in the C programming language). These assumptions are expressed with the following Tspecs:*

$$\begin{aligned} T^{Root} = T_2^1 = T_2^2 = T_3^1 &= \{f_1, f_2, f_3, f_4, f_6, f_7\} \\ T_1^1 &= \{f_5\} \\ T_1^2 = T_1^3 &= \{f_3, f_4, f_6\} \end{aligned}$$

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 *Fspecs*, which further restrict the space of trees.

Definition 2 *Define the following* Fspecs *(semantic constraints):*

1. F^{Root} – *the set of functions disallowed at the Root.*

2. $F_* - F_i$ *is the set of functions disallowed as direct callers to* f_i *(generally, a function is unaware of the caller; however, GP constructs a tree).*

3. $F_*^* - F_i^j$ *is the set of functions disallowed as* arg_j *to* f_i.

Example 2 *Continue example 1. Assume that we know that the sensor reading function f_4 does not provide the solution to our problem. We also know that boolean (generated by f_5) cannot be the answer (this information is actually redundant as it can be inferred from Tspecs; however, it will be easier for the user if no specific requirements are made as to how to specify non-redundant constraints). Also assume that for some semantic reasons we wish to exclude f_3 from calling itself (e.g., this is the integer-part function, which yields identity when applied to itself). These constraints are expressed with the following Fspecs (the other sets are empty):*

$$\begin{aligned} F^{Root} &= \{f_4, f_5\} \\ F_3 &= \{f_3\} \end{aligned}$$

3.2 Transformation of the constraints

3.2.1 Normal form

The above *Tspecs* and *Fspecs* provide a specification language for expressing problem constraints. Obviously this language is limited in power. However, it is quite useful (as our experiments illustrate - section 4) and we believe the expressible constraints are the most general that could be implemented without increasing the computational complexity of *lil-gp* (section 3.2.5 and 3.2.7).

Because *Tspecs* and *Fspecs* allow redundant specifications, an obvious issue is that of the existence of sufficiently minimal specifications. It turns out that after certain transformations, only a subset of *Tspecs* and *Fspecs* is sufficient to express all such constraints. This observation is extremely important, as it will allow efficient constraint enforcement mechanisms after some initial preprocessing. The first step is to extend *Fspecs*.

Proposition 1 *The following are valid inferences for extending Fspecs from Tspecs:*

$$\forall_{f_k \in F}(f_k \notin T_i^j \rightarrow f_k \in F_i^j)$$
$$\forall_{f_k \in F}(f_k \notin T^{Root} \rightarrow f_k \in F^{Root})$$

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

The above proposition is very important as it states that *Tspecs* can be expressed with *Fspecs*.

Definition 3 *If Fspecs explicitly satisfy proposition 1 then call them* T-extensive *Fspecs. If Fspecs do not satisfy proposition 1 for any function, then call them* T-intensive *Fspecs.*

In other words, *T-intensive Fspecs* list only semantics-based constraints which cannot be inferred from data types.

Proposition 2 *T-extensive Fspecs are sufficient to express all Tspecs.*
:: Consider function f_k and function f_i of type I (with arguments). Two cases are possible:

- $\neg(f_k \Rightarrow f_i^j)$. *Then $f_k \notin T_i^j$ in Tspecs, and according to proposition 1 $f_k \in F_i^j$. Thus, Fspecs express the same information that f_k cannot be called from the j^{th} argument of f_i.*
- $f_k \Rightarrow f_i^j$. *Then $f_k \in T_i^j$ in Tspecs. Thus, based on Tspecs there is no reason to exclude from being called by the j^{th} argument of f_i. However, Fspecs list additional constraints which supersede those of Tspecs. Thus, if $f_k \in F_i^j$ then f_k should be excluded regardless of Tspecs, and if $f_k \notin F_i^j$ then Fspecs and Tspecs say the same.*

Now we look at redundancies among *Fspecs.*

Proposition 3 *Suppose $f_k \in F$ and Fspecs 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 f_k cannot call f_i, then f_i will never be called by f_k on any of its a_k arguments.

However, F_* and F_*^* are not equivalent - a function may be allowed on some arguments but not on others. Nevertheless, both are not needed either - F_*^* *Fspecs* are stronger.

Definition 4 *If Fspecs explicitly satisfy proposition 3 then call them* F-extensive *Fspecs. Dropping all F_* from the F-extensive Fspecs gives* F-intensive *Fspecs.*

Proposition 4 *T-extensive F-intensive Fspecs are sufficient to express all Fspec constraints.*
:: Follows from proposition 3.

Definition 5 *Call the T-extensive F-intensive Fspecs the* normal form*. That is, the normal form contains only the F^{Root} and F_*^* Fspecs (after proper transformations).*

Proposition 5 *The normal form is sufficient to express all constraints of the Tspec/Fspec language.*
:: According to proposition 2 T-extensive Fspecs express or supersede Tspecs. According to proposition 4, T-extensive F-intensive Fspecs express all the same info as any other form of Fspecs.

It is not shown here, but the normal form is also the minimal form that expresses the *Tspec* and *Fspec* constraints.

Example 3 *Constraints of examples 1 and 2 have the following normal form:*

$$\begin{aligned} F^{Root} &= \{f_4, f_5, f_6\} && (1)\\ F_1^1 &= \{f_1, f_2, f_3, f_4, f_6, f_7\} && (2)\\ F_1^2 = F_1^3 &= \{f_1, f_2, f_5, f_7\} && (3)\\ F_2^1 = F_2^2 &= \{f_5\} && (4)\\ F_3^1 &= \{f_3, f_5\} && (5) \end{aligned}$$

The normal form expresses constraints in a unique and minimal form. These transformed constraints are consulted by *lil-gp* to restrict the search space. Obvious questions remain: how can crossover/mutation use the information in an efficient and effective way?. We propose to express the normal form differently – in *mutation sets* – to facilitate efficient consultations. In fact, we show that the overall $\underline{O}$ complexity for constrained mutation/crossover remain the same.

3.2.2 Useless functions

Given a specific set of constraints, it may happen that the constraints prohibit some functions from being used in any valid program - such functions would invalidate any program regardless of their position in the program tree. Detection of such cases is addressed in this section. Notice that this issue would be dealt with by *CGP lil-gp* itself since no node would be labeled with such a function. Early detection is rather a tool aimed at presenting such situations to the user.

Definition 6 *If a function from F cannot label any nodes in a valid tree, call it a* useless *function.*

Proposition 6 *A function $f_i \in F$ is useless iff*

- *it is a member of all sets of the normal form, or*
- *it is a member of all sets of the normal form except for only sets associated with useless functions.*

:: F_*^* sets of the normal form list functions excluded from being called as children of other functions. F^{Root} lists functions excluded from labeling the Root. A function excluded from labeling the Root and excluded from labeling all children nodes cannot possibly label any node in a valid program. On the other hand, if the function does not appear in at least one of the sets of the normal form, then it can indeed label some nodes. The only exception to the latter is when the function is allowed to be directly called only from other functions which are found to be useless. Because the useless functions cannot label any nodes, then the function in question will never be called in any tree.

Proposition 7 *Removing useless functions from F does not change the CGP search space. That is, exactly the same programs can be evolved before and after the removal.*

:: Useless functions cannot appear in any valid tree.

3.2.3 Mutation sets

lil-gp allows parameters determining how deep to grow a subtree while in mutation. That is, *lil-gp* allows differentiation between functions of type I and terminal nodes (labeled with type II or III). We need to provide for the same capabilities.

Definition 7 *Define $\mathcal{F}_N$ to be the set of functions of type I that can label (thus, excluding useless functions) node N without invalidating an otherwise valid tree containing the node. Define $\mathcal{T}_N$ to be the set of terminals T that can label node N the same way.*

Proposition 8 *Assume the normal form for constraints, and node N, not being the Root and being the j^{th} child of a node labeled f_i. Then*

$$\begin{aligned}\mathcal{T}_N &= \{f_k | f_k \notin F_i^j \wedge f_k \in F_{II} \cup F_{III}\} \\ \mathcal{F}_N &= \{f_k | f_k \notin F_i^j \wedge f_k \in F_I\}\end{aligned}$$

:: The normal constraints express all Tspecs and Fspecs according to proposition 5. N is not the Root, so it must be a child of a node labeled with functions with arguments. F_i^j in the normal form lists all functions excluded from labeling the child N.

Proposition 9 *Assume the normal form for constraints and node N being Root. Then*

$$\begin{aligned}\mathcal{T}_N &= \{f_k | f_k \notin F^{Root} \wedge f_k \in F_{II} \cup F_{III}\} \\ \mathcal{F}_N &= \{f_k | f_k \notin F^{Root} \wedge f_k \in F_I\}\end{aligned}$$

::Arguments follow those for Proposition 8.

Definition 8 *Let us denote $\mathcal{T}_{Root}$ and $\mathcal{F}_{Root}$ the pair of mutation sets associated with Root. Let us denote $\mathcal{T}_i^j$ and $\mathcal{F}_i^j$ the pair of mutation sets for the j^{th} child of a node labeled with f_i.*

Proposition 10 *For an application problem, there are $1 + \sum_{i=1}^{|F_I|}(a_i)$ mutation set pairs.*

:: There is exactly one pair for Root. For other nodes, the mutation sets are determined by what function labels the parent node, and which child of the parent the node is. Parent nodes are of type I. If the label of the parent is f_i, then it can have exactly a_i different children.

The above implies that the information expressed in the normal form can be expressed with $2 \cdot (1 + \sum_{i=1}^{|F_I|}(a_i))$ different function sets, while only two sets (one pair) are needed in *lil-gp* itself. Now we show how these sets alone are sufficient to initialize *CGP lil-gp* programs with only valid trees, to mutate valid trees into valid trees, and to crossover valid trees into valid trees.

Proposition 11 *For any non-root node N of a valid program at least one of the two mutation sets is guaranteed not to be empty. The same is true for Root (see proposition 12).*

:: Suppose N is labeled with f_i. If N is an internal node, then $f_i \in \mathcal{F}_N$. If N is a leaf, then $f_i \in \mathcal{T}_N$.

Example 4 *Here are selected examples of mutation sets generated for example 3:*

$$\begin{aligned} \mathcal{T}_{Root} &= \{f_7\} \\ \mathcal{F}_{Root} &= \{f_1, f_2, f_3\} \\ \mathcal{T}_3^1 &= \{f_4, f_6, f_7\} \\ \mathcal{F}_3^1 &= \{f_1, f_2\} \end{aligned}$$

3.2.4 Constraint feasibility

Unfortunately constraints may be so severe that only empty or only infinite trees are valid. In the first case, GP would fail to initialize trees (or it would try infinitely). In the second case, GP would run out of memory or it would fail, as in the first case, if size restrictions were imposed. To avoid such problems, this could be detected early and the troublesome functions can be identified and possibly removed from the function set. We exclude the empty tree from being valid. In other words, a tree must contain at least one node to constitute a potential solution.

Proposition 12 *If $(\mathcal{T}_{Root} = \emptyset) \wedge (\mathcal{F}_{Root} = \emptyset)$ then no valid trees exist.*

:: There is no way to label the Root. Thus, valid (nonempty) trees do not exist. Stated differently, a valid tree cannot have both of these sets empty.

Proposition 12 identifies trivial cases when no valid trees exist because only empty trees are valid. However, a more common problem might be that only infinite trees exist.

Example 5 *Consider a function $f_i \in F_I$ such that $\exists_{j \in a_i}(F_i^j = F \setminus \{f_i\}$ in the normal form. In other words, on the given argument the function can only call itself. It can be verified that any node N being the j^{th} child of any node labeled with f_i will have the following mutation sets (proposition 8): $\mathcal{T}_N = \emptyset$ and $\mathcal{F}_N = \{f_i\}$. This means that the child node can only be labeled the same way as the parent: f_i. Recursively, the same will apply to its j^{th} child. Note that the same can happen through indirect recursion as well.*

Definition 9 *We say that a subtree whose root is labeled f_i* can be finitely instantiated without- *(*cbfiw-*)$F' \subseteq F$ iff finite valid trees without labels from F' do exist.*

Proposition 13 *A tree with its root labeled f_i cannot be finitely instantiated without functions from F') (f_i $cbfiw - F'$) iff either is true*

$$\exists_{j \in a_i} (\mathcal{T}_i^j = \emptyset \wedge \mathcal{F}_i^j = \{f_i\})$$

$$\exists_{j \in a_i} (\mathcal{T}_i^j = \emptyset \wedge \forall_{f_k \in \mathcal{F}_i^j \setminus (F' \cup \{f_i\})} \neg (f_k \ \mathit{cbfiw} - (F' \cup \{f_i\}))$$

:: If the sets $\mathcal{T}_i^*$ are nonempty for all a_i children, all the children can be instantiated to leaves. Any child having $\mathcal{T}_i^j = \emptyset$ and only allowing recursive calls (first case) will necessarily create an infinite tree. Any such child which also allows other type I function calls must be eventually instantiated with a finite tree - only indirect recursion would obviously lead to infinite trees - those are excluded in the second case.

Proposition 13 helps identify functions causing only infinite trees to be valid. Such cases can be reported to the user. Moreover, the troublesome functions can be removed from consideration.

This feature, along with useless functions, is not currently implemented. Montana [10] presents a procedure which also takes tree depth into account.

3.2.5 *CGP lil-gp* mutation

lil-gp mutates a tree by selecting a random node (different probabilities for internal and external nodes). The mutated node becomes the root of a subtree, which is grown as determined by some parameters. To stop growing, a terminal function is used as the label. To force growing, a type I function is used as the label. In *CGP lil-gp* the only difference is that a subset of the original functions provides candidates for labels.

Operator 1 (Mutation) *To mutate a node N, first determine the kind of the node (either Root, or otherwise what the label of the parent is and which child of that parent N is). If the growth is to continue, label the node with a random element of $\mathcal{F}_N$ and continue growing the proper number of subtrees, each grown recursively with the same mutation operator. Otherwise, select a random element of $\mathcal{T}_N$, instantiate it if from F_{III}, and stop expanding N. If growing a tree and $\mathcal{F}_N = \emptyset$, then select a member of $\mathcal{T}_N$ (guaranteed not to be empty under proposition 11). If stopping the growth and $\mathcal{T}_N = \emptyset$, then select a member of $\mathcal{F}_N$ (this will unfortunately extend the tree, but it is guaranteed to stop according to proposition 13).*

Proposition 14 *If a valid tree is selected for mutation, operator 1 will always produce a valid tree. Moreover, this is done with only constant overhead.*

:: The mutation sets express exactly the same information as Tspecs and Fspecs. Moreover, the only implementation difference is to consult one of $1 + \sum_{i=1}^{|F_I|}(a_i)$ instead of a single set of function labels in lil-gp. Which set to consult is immediately determined from the parent node and can be accessed in constant time given proper data structure.

Example 6 *Assume the mutation sets of example 4. Assume mutating parent1 as in figure 1. Assume the node N is selected for mutation. It is the 1st child of a node labeled with* f_3. *Thus,* $\mathcal{T}_N = \mathcal{T}_3^1 = \{f_4, f_6, f_7\}$ *and* $\mathcal{F}_N = \mathcal{F}_3^1 = \{f_1, f_2\}$. *If the current mode is to grow the tree, then the mutated node will be randomly labeled with either* f_1 *or* f_2. *If the current node is to generate a leaf, then label N with either* f_4, f_6, *or* f_7.

3.2.6 *CGP lil-gp* initialization

Operator 2 (Create a valid tree) *To generate a valid random tree, create the Root node, and mutate it using the mutation operator.*

Proposition 15 *If* $\mathcal{T}_{Root} \neq \emptyset \vee \mathcal{F}_{Root} \neq \emptyset$ *and functions such that trees with such roots cannot* $cbfiw - \emptyset$ *are removed from* F *then operator 2 will create a tree with at least one node, the tree will be finite and valid with respect to constraints.*

:: Because of the conditions at least one node can be labaled. The functions remaining in the mutation sets can label trees with finite elements and guarantee validity.

3.2.7 *CGP lil-gp* crossover

The idea to be followed is to generate one offspring by replacing a selected subtree from *parent1* with a subtree selected from *parent2*. To generate two offspring, the same may be repeated after swapping the parents.

Operator 3 (Crossover) *Suppose that node N from parent1 is selected to receive a material from parent2. First determine* $\mathcal{F}_N$ *and* $\mathcal{T}_N$. *Assume that* F_2 *is the set of labels appearing in parent2. Then,* $(\mathcal{F}_N \cup \mathcal{T}_N) \cap F_2$ *is the set of labels determining which subtrees from parent2 can replace the subtree of parent1 starting with N. In other words, any subtree of parent2 whose root is labeled with one of* $(\mathcal{F}_N \cup \mathcal{T}_N) \cap F_2$ *can replace N and still generate a valid tree.*

Proposition 16 *If two valid trees are selected for crossover, the operator will always produce a valid tree. Moreover, this is done with only the same (order) computational complexity.*

:: For the first part, arguments follow those of proposition 14 since crossover is based on the same mutation sets. Crossover is implemented in lil-gp in such a way that a random number up to the number of nodes in parent2 is generated, and then the tree is traversed until the numbered node is encountered (can be done separately for internal and external nodes). Therefore, crossover's complexity is in the size of the tree $\underline{O}(n)$.

CGP lil-gp does not know ahead of the traversal how many nodes will be found applicable. During the traversal, applicable nodes are indexed for constant-time access and they are counted. At the end of the tree, a random number up to the counter is generated, and the proper node is immediately accessed. On average, this requires traverasl twice as long, but

in the same order.
Instead of generating indexed constant-time-access structures, another traversal may follow. This does not change the overall complexity (adds another $\underline{O}(n)$).

< Insert Fig1 >

Figure 1: Illustration of mutation and crossover.

Example 7 *Assume mutation sets of example 4. Assume parent1 and parent2 as in figure 1. Assume the node N is selected for replacing with a subtree of parent2. It is the 1st child of a node labeled with f_3. Then, $\mathcal{T}_N = \mathcal{T}_3^1 = \{f_4, f_6, f_7\}$ and $\mathcal{F}_N = \mathcal{F}_3^1 = \{f_1, f_2\}$, and only the subtrees with the shaded roots can be used to replace N. Crossover would select a random element from a so marked set of nodes, and move the corresponding subtree.*

4 Illustrative Experiment

In this section, we follow a practical example intended to illustrate how problem-specific knowledge can be used to come up with various constraints. In this experiment, we explore different constraints that can be used to express, and thus restrict, some of the redundant solutions from being explored by GP. This is intended as illustration, but we also explore the implications of such restrictions on the behavior of GP. In fact, this issue arises in any state-space search, and has yet to be addressed. We hope that our tool will help in studying this issue.

We will use the widely studied 11-multiplexer problem [8]. Multiplexer is a boolean circuit with a number of inputs and exactly one output. In practical applications, a multiplexer is used to propagate exactly one of the inputs to the output. For example, in the computer CPU (central processing unit) multiplexers are used to pass binary bits (via a group of multiplexers) from exactly one location (*e.g.*, one register) to the ALU (arithmetic-logic unit). Multiplexer has two kinds of binary inputs: address and data. The address combination determines which of the data inputs propagates to the output. Thus, for a address bits, there are 2^a data bits. 11-multiplexer has 3 address and 8 data bits. Let us call them $a_0 \dots a_2$ and $d_0 \dots d_7$, respectively. For example, when the address is 110 (the boolean formula $a_2 a_1 \overline{a_0}$), then d_6 is passed to the output.

11-multiplexer implements a boolean function, which can be expressed in DNF (disjunctive normal form) as:

$$a_2a_1a_0d_7 \vee a_2a_1\overline{a_0}d_6 \vee a_2\overline{a_1}a_0d_5 \vee a_2\overline{a_1}\,\overline{a_0}d_4 \vee \overline{a_2}a_1a_0d_3 \vee \overline{a_2}a_1\overline{a_0}d_2 \vee \overline{a_2}\,\overline{a_1}a_0d_1 \vee \overline{a_2}\,\overline{a_1}\,\overline{a_0}d_0$$

In [8], Koza has proposed to use the following function set $F_I = \{and, or, not, if\}$ and terminals $F_{II} = \{a_0 \dots a_2, d_0 \dots d_7\}$ (no F_{III} functions) for evolving the 11-multiplexer function with GP. In this case, GP evolves trees which are labeled with the above primitive elements, each element having the standard interpretation. The only feedback to this evolution is the evaluation (environment), which assigns a fitness value to each tree based on the number of the possible 2048 input combinations which compute the correct output bit.

The function set is obviously complete, thus satisfying *sufficiency*. However, the set is also redundant - a number of type I subsets, such as $\{and, not\}$, are known to be sufficient to represent any boolean formula. Thus, by placing restrictions on function use, we may reduce the amount of redundant subspaces in the representation space. However, we do not know what function sets make it easier, or more difficult, to solve this problem by evolution. In fact, the following experiments will spark very interesting observations suggesting that *sufficiency* itself is not strong enough to predict learning properties - in addition to providing the necessary functions/terminals, one should also provide "the right" functions/terminals.

As to *closure*, it is trivially satisfied for this problem since all terminals (address/data sensors) and all type I functions return boolean. Thus, this problem does not have any invalid subspaces - all constraints will be used only to reduce the number of redundant/undesired trees. Even given this triviality, it is a very interesting problem.

We set a number of experiments, intended to illustrate how *CGP lil-gp* can be used to utilize various constraints, drawn from problem-specific knowledge. For each case, we repeat and average 10 independent runs, with a population of 2000, 0.85/0.1/0.05 probabilities for crossover/selection/mutation, and otherwise the default parameters. We report averages of best solutions generated at 5-iteration increments (discrete learning curves) while running for 100 iterations.

Previously, we observed that the constraint language allows redundant specifications. In fact, many of the constraints we subsequently use can be expressed in a number of different ways (it is the translator that generates unique equivalent constraints). To make the presentation more systematic, we assume that *Tspecs* stay constant, and all constraints are expressed with *Fspecs*. In one case, however, we illustrate how the same constraints can be expressed with different *Tspecs*. The generic *Tspecs* we use do not impose any constraints. Thus,

$$T^{Root} = T^*_* = \{and, or, not, if, a_0 \ldots a_2, d_0 \ldots d_7\}$$

where '*' indicates any possible value, here meaning that all sets are the same.

4.1 Unconstrained 11-multiplexer with *lilgp* (base experiment)

Even though it is not our current intention here to evaluate the impact that the reduction of redundant subspaces may have on search properties, we set a benchmark obtained from unconstrained *lil-gp* on the same problem. In this experiment, we evolve 11-multiplexer solutions using the above function set and no constraints. Thus, we recreate Koza's experiments, except that we use *lil-gp* (and not *CGP lil-gp* either).

The remaining experiments all use *CGP lil-gp*.

4.2 Experiment E_0: unconstrained 11-multiplexer with *CGP lil-gp*

This is the same base experiment except that it is run with *CGP lil-gp*. Thus, there are no constraints (all *Fspecs* are empty). This experiment may be treated as informal validation - formal validation/verification is done separately and will be reported elsewhere.

4.3 Experiment E_1: using sufficient set $\{and, not\}$

We observe that $\{and, not\}$ is a sufficient type I function set. Thus, we run an experiment with only these two type I functions. While in this specific case it is also possible to run *lil-gp* with only these functions (by modifying and then recompiling the program), this is not our objective. Instead, we show how this particular constraint can be presented in *CGP lil-gp*. Our constraint is that of the four type I functions, if and *or* be not used at all. This can be expressed with the following *Fspecs*:

$$F^{Root} = F^*_* = \{if, or\} \qquad F_* = \emptyset$$

We should note that even though $\{and, not\}$ is a sufficient set, the 11-multiplexer function expressed with these two functions is necessarily more complex. Thus, we should not expect any payoff from this constraint (this is another example of problem-specific knowledge). In other words, we suspect that this is not "the right" sufficient set.

4.4 Experiment E_2: DNF

We attempt to generate DNF (disjunctive normal form) solutions. Obviously, if must be excluded. However, this is not sufficient. We must also ensure that *or* is distributed over *and*, and that *not* applies to type II functions (atoms) only. This can be expressed (one of possible options) with the following *Fspecs*:

$$\begin{aligned} F^{Root} &= \{if\} & F_* &= \emptyset \\ F^*_{if} &= \emptyset & F^*_{not} &= \{if, or, and, not\} \\ F^*_{and} &= \{if, or\} & F^*_{or} &= \{if\} \end{aligned}$$

4.5 Experiment E_3: structure-restricted DNF

The above DNF specification leaves many interpretation-isomorphic trees. In this experiment, we intend to remove some of those redundancies (though not all). We constrain the trees to grow conjunctions and disjunctions to the left only (thus, we prohibit right-recursive calls on *or* and *and*). This is accomplished with the following modifications to *Fspecs* of E_2:

$$\begin{aligned} F^{Root} &= \{if\} & F_* &= \emptyset \\ F^*_{if} &= \emptyset & F^*_{not} &= \{if, or, and, not\} \end{aligned}$$

$$F^1_{and}\{if, or\} \qquad F^2_{and} = \{if, or, and\}$$
$$F^1_{or}\{if\} \qquad F^2_{or} = \{if, or\}$$

Previous experience with other evolutionary algorithms using DNF representation suggest that DNF is "the right" representation (GIL system, [5]). Thus, we would expect both E_2 and E_3 to do relatively well. We will shortly observe that (and speculate why) this is not the case.

4.6 Experiment E_4: using $\{if\}$ only

Here we observe that the type I function set $F_I = \{if\}$ is completely sufficient for the task of learning the 11-multiplexer. Even though studying that is not our explicit objective, we may compare the learning characteristics of this experiment with those of other complete function sets (E_1, E_2, and E_3), giving us some insights as to what functions make it easier for GP to evolve solutions to the 11-multiplexer problem. Our observations will be rather striking.

Restricting trees to use this function only can be accomplished with the following *Fspecs*:

$$F^{Root} = F^*_{if} = \{and, or, not\} \quad F_* = \emptyset$$
$$F^*_{or} = F^*_{and} = F^*_{not} = \text{irrelevant}$$

4.7 Experiment E_5: E_4 with problem-specific knowledge

Now, suppose that in addition to observing that $\{if\}$ is a sufficient type I function set we also use some additional problem-specific knowledge. For example, suppose we know that the first three bits are addresses and the others are data bits. Knowing the interpretation of if (which we do since we implement it), we may further conclude that the condition argument (#1) should test addresses, and the other arguments should compute and thus return data bits. This constraint could be completely expressed with a slightly enlarged function set. To avoid extra complexity, we express a somehow lesser constraint, one which restricts only immediate arguments (in the original theory it is possible to specify the stronger constraint for this function set, because that theory is based on sets rather than functions [6]). This can be expressed with the following *Fspecs*:

$$F^{Root} = \{and, or, not, a_0, a_1, a_2\} F_* = \emptyset$$
$$F^1_{if} = \{and, or, not, d_0 \ldots d_7\}$$
$$F^2_{if} = F^3_{if} = \{and, or, not, a_0, a_1, a_2\}$$
$$F^*_{or} = F^*_{and} = F^1_{not} = \text{irrelevant}$$

or the same *Fspecs* as those of E_4 plus the following *Tspecs* (this is just for illustration; however, as indicated earlier, *Tspecs* are intended to restrict *closure*):

$$\begin{aligned} T^1_{if} &= \{if, a_0, a_1, a_2\} \\ T^{Root} = T^2_{if} = T^3_{if} &= \{if, d_0 \ldots d_7\} \end{aligned}$$

4.8 Experiment E_6: E_5 with further heuristic knowledge

Further suppose that we prevent trees of E_5 from using if on its first argument. This further reduces redundancy, while still allowing solutions to evolve. This can be accomplished with:

$$F^{Root} = \{and, or, not, a_0, a_1, a_2\} F_* = \emptyset$$
$$F^1_{if} = \{and, or, not, if, d_0 \ldots d_7\}$$
$$F^2_{if} = F^3_{if} = \{and, or, not, a_0, a_1, a_2\}$$
$$F^*_{or} = F^*_{and} = F^1_{not} = \text{irrelevant}$$

4.9 Experiment E_7: E_6 relaxed

Finally, suppose that we want to allow another function to enrich our explored search space – *not* to be used in the condition part of if. However, we make sure that it only applies to non-negated address bits. This of course introduces additional redundancy. This can be accomplished with:

$$F^{Root} = \{and, or, not, a_0, a_1, a_2\} F_* = \emptyset$$
$$F^1_{if} = \{and, or, if, d_0 \ldots d_7\}$$
$$F^2_{if} = F^3_{if} = \{and, or, not, a_0, a_1, a_2\}$$
$$F^1_{not} = F \setminus \{a_0, a_1, a_2\}$$
$$F^*_{or} = F^*_{and} = \text{irrelevant}$$

With the above, E_7 will evolve solutions of the form illustrated in figure 2.

< Insert Fig2 >

Figure 2: Solution form for E_7.

4.10 Experimental results and discussion

The results are very interesting, some even striking. To illustrate them, we present two figures. Figure 3 presents quality of the best solutions captured in 5-iteration intervals (averaged over 5 independent runs). In cases when a run finds the perfect (2048) tree before the 100th iteration, its 2048 evaluation is used for averaging in subsequent iterations.

Figure 4 presents complexity, measured by the number of nodes, of the same best trees. For each run which completes before the 100th iteration, complexity 0 is used for averaging on subsequent generations. This way the curves are directly proportional to average time needed to evaluate an individual (since no more work is necessary after a solution is found). In other words, lower complexity would result in lower processing times per generation. Moreover, the area bounded by each curve is directly proportional to the total time needed for evolution (with a bound 100 iterations).

First, when constraints are not present (base and E_0), both *lil-gp* and *CGP lil-gp* perform very similar searches (discrepancies result from a different number of random calls, thus resulting in stochastically different runs). As indicated before, this is not intended to serve as verification/validation. More systematic experiments are used to accomplish that, with extra processing to ensure the same random calls take place - in which case both programs explore exactly the same trees. Because the runs were very similar here, figures 3 and 4 report averages from these two experiments.

Forcing evolution with $\{and, not\}$ type I set (E_1), even though it dramatically reduces the number of redundant solutions being explored, has a disastrous effect. It seems that the most important reason for this degradation is that, as pointed out shortly, if is extremely efficient in solving this problem with GP. Moreover, 11-multiplexer expressions using $\{and, not\}$ are necessarily more complex. This would require extra processing to evolve - as seen in Figure 3, the learning curve has not saturated after 100 iterations.

Forcing DNF functions to evolve (E_2) has equally disastrous effects on the program. In this case, even further restrictions on tree structures (E_3) failed to compensate for the disadvantage. It seems that the reasons are similar to those above - if will prove to be the most effective and thus extremely important. The fact that GP fails to efficiently evolve DNF solutions is striking when compared against another evolutionary program designed for machine learning. GIL [5] is a genetic algorithm with specialized DNF representation, specialized inductive operators, and evolutionary state-space search controlled by inductive heuristics. In reported experiments, while evolving solutions to the same function, but in a more challenging environment in which only 20Our DNF GP evolved less than 90Even though a direct comparison was not an objective here, one may draw some conclusions. In this case, both programs were using the same representation (DNF). The only difference is that *CGP lil-gp* used only blind crossover/mutation, fired with static probabilities, while GIL used operators modeling the inductive methodology, whose firing was controlled by heuristics. This suggests that such problem-specific knowledge is extremely important to evolutionary problem solving.

< Insert Fig3 >

Figure 3: Comparison of the quality of the best-of-population tree.

Because of similar results, figures 3 and 4 report averages of E_1, E_2, and E_3.

In the other experiments we investigate the utility of the if function for this specific problem. The reason for this experiment is that our previous results with restricted but still sufficient function sets failed to improve search characteristics, instead degrading the performance and leading to our suspicion that this interpretation-rich function is extremely important for solving this problem with GP. Thus, E_4 was set to evolve with only one type I function: if. Results are strikingly obvious: perfect solutions finally emerge from this evolution, on the average after about 70 iterations. However, time complexity increases due to the increase in tree sizes (figure 4).

Increased tree sizes translate directly into longer processing time per iteration. Thus, the wall-clock performance might not necessarily improve. To alleviate the problem, we used additional problem-specific information about different interpretation of address and data bits (E_5). This leads not only to further speed up in evolution (figure 3). The evolving trees also have the smallest sizes from among all experiments (figure 4). This result supports our previous conjecture that problem-specific knowledge is crucial here. It also illustrates how the generic *CGP lil-gp* can utilize this kind of information (GIL, on the other hand, was designed and implemented with such problem-specific knowledge from the beginning).

In other words, this result indicates that it is indeed important to provide "the right" and minimal set of functions for GP. For example, comparing results from E_0 and E_4 one may see a dramatic improvement despite the fact that both experiments use the identified if function. This indicates that reducing the redundant subspace pays off in this case, but only because "the right" subspace was pruned away.

< Insert Fig4 >

Figure 4: Comparison of complexity needed for evolving solutions in 100 generations (complexity 0 used on finished runs).

Finally, providing additional heuristic about the desired solutions, and thus pruning away other otherwise valid solutions, leads to even better speed ups (E_6 and E_7 in figure 3 are averaged since they produced indistinguishable curves). This further supports our observation that providing such information is advantageous not only to generate solutions with some specific characteristics but to speeding up evolution as well. Unfortunately, usually this can only be done by a careful redesign of the algorithm/representation/operators, or the function set in GP. In CGP, no changes are needed.

Between E_6 and E_7 it is worthwhile to point out that E_6, which uses less redundant search space, explores trees of slightly lower complexity. Finally, between the two and E_5, it is interesting to observe that while the former evolve perfect solutions in many fewer generations, this involves trees of larger sizes. In fact, in terms of clock-time performance, E_5 outperforms these two (areas in figure 4).

5 Summary

This paper describes a method to prune constraints-identified subspaces from being explored in GP search. The constraints are allowed in a user-friendly language aimed at expressing syntax and semantics-based restrictions to *closure*. Specific constraints lead to the exclusion of syntactically invalid, redundant, or simply undesired trees from ever being explored. Such pruning may not only lead to more efficient problem solving with *lil-gp*. When studied systematically, it may also give insights about pruning redundant subspaces from any state-space search.

We have presented a complete methodology and illustrated it with an example. We have also used the 11-multiplexer problem to illustrate practical application of the methodology. Even though illustration was our primary goal, some interesting observations were made.

It has been obvious that the function set proposed by Koza for solving this problem is redundant. Our experiments suggest that reducing those redundancies, and thus reducing the search space, is not necessarily advantageous. However, if "the right" choices are made, a tremendous payoff can be expected. This is further amplified by using additional problem-specific knowledge. *CGP lil-gp* allows us to express such information with a generic constraint language, alleviating the need for devising specialized representation/operators. However, by comparing the results with those of another specialized algorithm, we may observe that such a specialized algorithm makes it advantageously possible to implement other problem-specific information and heuristics.

In the future, we plan to make more systematic testing aimed at supporting the observations made here. In particular, we did not even explore the methodology's impact on the more serious problem of invalid subspaces, where we expect the benefits to amplify. We are also currently extending the implementation for ADFs (automatically defined functions), which will allow similar capabilities to Montana's *generic functions* [10] yet more general (as our crossover is more general).

One should point out that the current constraint specification language does not allow for arbitrary constraints to be expressed. In particular, this *lil-gp*'s version is even weaker than the originally proposed methodology. Thus, for the future we also plan to explore extending the language and/or this implementation of *lil-gp*.

References

[1] Leonard Bolc & Jerzy Cytowski. *Search Methods for Artificial Intelligence*. Academic Press, 1992.

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

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

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

[5] Cezary Z. Janikow. *"A Knowledge-Intensive GA for Supervised Learning"*. *Machine Learning* 13 (1993), pp. 189-228.

[6] Cezary Z. Janikow. *"Constrained Genetic Programming"*. Submitted to *Evolutionary Computation*.

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

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

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

[10] David J. Montana. *"Strongly typed genetic programming"*. *Evolutionary Computation*, Vol. 3, No. 2, 1995.

[11] Douglas Zonker & Bill Punch. *lil-gp 1.0 User's Manual*. zonker@isl.cps.msu.edu.