Cezary Z. Janikow

Dept. of Mathematics and Computer Science
University of Missouri – St. Louis

Evolutionary Learning with Constrained Genetic Programming

1. Introduction

Solving a problem on the computer involves two elements: representation of the problem, 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. However, separating them has proven advantageous. This idea has been long 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), proposed by Holland (1975), use a population of chromosomes coding individual potential solutions. These chromosomes undergo simulated evolution facing Darwinian selective pressure. Chromosomes which are better with respect to a simulated environment (the problem being solved) have increased survival chances (*reproduction*). Chromosomes interact with each other via *crossover* to produce new offspring solutions, and they are subjected to *mutation*. Traditional genetic algorithms operate on fixed-length chromosomes, which may not be suitable for some problems. To deal with that, some genetic algorithms adopted variable-length representation, as in machine learning: Goldberg (1989), Janikow (1993). Moreover, traditional GAs use low-level binary representation, but many recent applications use more abstracted alphabets: Davis (1991).

Genetic programming (GP), proposed by Koza (1992, 1994), uses treesto represent individuals: also see Kinnear. At first used to generate LISP computer programs, GP is also being used to solve problems where solutions are arbitrary structures: Koza (1994). Tree representation is richer than that of linear fixed-length strings. This allows for representing (functional) computer programs or other variable-size structures. However,many sought solutions have more or less severe limitations (called here *constraints*) on the context in which elementscan appear. To avoid these constraints, GP operates under *closure:* Koza (1992), which is accomplished through extended interpretation as needed. Because of the shortcomings of closure, Koza has proposed *structure-preserving crossover*, whose primary intention is to preserve structural constraints imposed by automatic modules (ADFs): Koza (1994).

For applicationswith rich domain constraints, two similar GP extensions have been proposed: Strongly Typed GP (STGP) and Constrained GP (CGP). Both follow the idea of *closing* the search in the feasible (or desired) space, originally proposed by Michalewicz and Janikow (1996) for GAs. In that approach, the initial population is generated so that each chromosome satisfies all constraints. Every operator (mutation and crossover) is guaranteed to produce constraint-valid offspring if constraint-valid parents are used.

STGP, proposed by Montana (1995), assigns types to functions, arguments, and terminals. The initial population is filled with valid structures, and every mutation and crossover preserves this validity. STGP has been used by many others to develop practical applications.

CGP has been at the same time developed at NASA/JSC, by Janikow, as a means to control the search space for GP applications in robotics: Janikow (1996). On the surface, it offers similar capabilities, with the major difference in its implementation - CGP is implemented into *lil-gp* (Zongker and Punch, on-line) and thus offers all capabilities available there (except for ADFs for the moment). However, deeper analysis reveals that CGP offers more by providing formal closure and complexity analysis, stronger crossover: Janikow (1996), as well as means to specify non-type-related constraints, heuristic constraints (Janikow, on-line), and evolution of representation (work in progress).

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 domains [1,100] would involve searching through the space of 10,000 solutions. This search may be random, enumerated exhaustive, or heuristic. However, in most practical problems of interest, the space of potential solutions is too large to be explicitly retained and/or 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. 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: Bolc and Cytowski (1992).

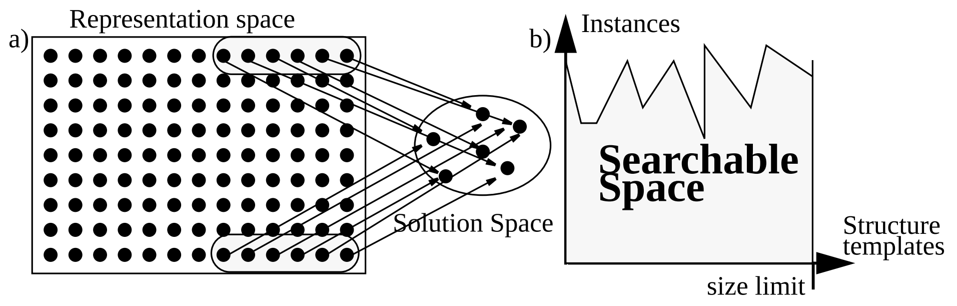

Figure 1 a) Search (representation) space and solution space of a state-space search. b) GP search space.

In such searches, an algorithm’s representation spans the search (representation) space. Usually the space is larger than the space of sought solutions. For example, in the above 2-dim integer problem, a binary GA would likely span 128x128 search space. In more complex problem, the size mismatch might be more severe. Then, a subspace of the space is sufficient to accomplish one-to-one mapping to the solution space (and thus to provide feasible search). Because exhaustive mapping is needed, redundancy (or sometimes invalidity) is introduced. This is illustrated in Figure 1a, where mappings from two redundant subspaces are shown. Recent findings suggest that the human DNA does indeed include redundant and/or unused genes, which may become activated through mutation. This has led some researchers to introduce such redundancies explicitly in a controllable fashion. Others claim such redundancies to be useful carriers of genetic information (*e.g., introns*, Koza 1992). However, heavily constrained applications, with many redundancies, researchers have proposed penalties, repair algorithms, and, if possible, operations closed in the feasible/desired space: Michalewicz and Janikow (1996).

GP’s search space is two-dimensional. The first dimension is that of unlabeled (uninstantiated) structures, which may be called *structure templates*. The second is that of *instances* of those templates. Assuming that structure templates are ordered by size, and that there is limit on that size, the resulting GP search space is illustrated as the shaded area of Figure 1b. CGP relies on identifying (implicitly, by constraints) both templates and instances which *will not* ever be explored due to closing the search in the remaining space. This corresponds to searching only a portion (yet sufficient) of the shaded area of Figure 1b.

3. CGP

CGP offers means to provide problem constraints similar to those of STGP (section 3.1), closed search with negligible implementational overhead (section 3.2), overloaded functions (section 3.3), and heuristic (weighted) constraints.

3.1 *T*- and *F*- constraint Specifications

CGP does not rely on explicit type specifications (unlike STGP). Instead, constraints are explicitly presented as sets of functions/terminals which are allowed (*Tspecs*) and disallowed (*Fspecs*) in a given context (function-argument). *Tspecs* are indented to provide type-based constraints, given explicitly for each function-argument. CGP uses the following *Tspecs*:

1. T^{Root} - the set of functions which return data type compatible with the problem specification. Therefore, this is the set of function/terminals which produce the correct type to be computed by a program.
2. T_i^j - the set of functions compatible with the *j*th argument of f_i.

   Example1 Assume three functions $F_I = \{f_1, f_2, f_3\}$ with arities 3, 2, and 1, respectively. Assume the terminal $F_{II} = \{f_4\}$ and ephemeral constants $F_{III} = \{f_5, f_6, f_7\}$. 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. The example assumes that integers are compatible with reals, while booleans are not compatible with either. Then

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

However, syntactic fit does not necessarily mean that a function *should* call another function. One needs additional specifications based (for instance) on domain semantics or heuristics. These are provided by means of *Fspecs*, which further restrict the space of trees.

3. F^{Root} - the set of functions disallowed at the Root.
4. F_i - the set of functions disallowed as direct callers to f_i
5. F_i^j - the set of functions disallowed as the *j*th argument of f_i.

Example2 Continue Example1. 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*). Also assume that for some reasons we wish to exclude f_3 from calling itself (*e.g.*, this is the *floor* function, which yields identity when applied to itself). These constraints are expressed with the following *Fspecs* (the other sets are empty): $F^{Root} = \{f_4, f_5\}$, $F_3 = \{f_3\}$

3.2 Normal Form, Mutations sets, and Closed Evolution

In Janikow (1996) we have shown (constructively) that constraints expressed with the above language can be uniquely represented in a minimal *normal form*. Such constraints can then be represented as *mutation sets* - to be used by closed evolution. Moreover, we have shown that given proper data structures (mutation sets), closed operators (which guarantee to produce constraint-valid offspring from constraint-valid parents) can be implemented with the same complexity (and negligible constant overhead) as the unconstrained operators of *lil-gp*. Except for the different means to specify constraints, and the resulting implications as mentioned above, this part of CGP is similar to STGP except for crossover - CGP's crossover is able to produce more constraint-valid trees than does STGP: Janikow (1996).

Example3 Here are selected examples of mutation sets generated for Example1 and Example2. $F_3^1 = \{f_1, f_2\}$ is the set of functions which can label the first child of a node labeled as f_3 without constraint-invalidating a structure. $T_3^1 = \{f_4, f_6, f_7\}$ is the set of such terminals.

.

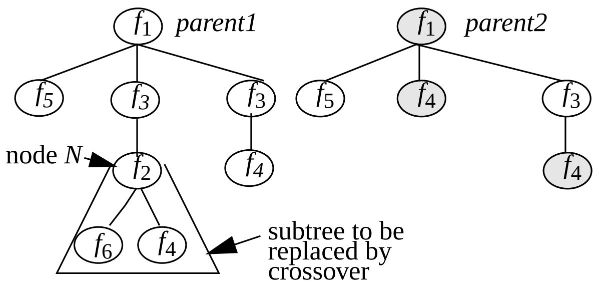

Figure 2 Illustration of mutation and crossover.

Example4 Assume mutation sets of Example3. Assume *parent1* and *parent2* as in Figure 2. 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, $T_N = T_3^1 = \{f_4, f_6, f_7\}$ and $F_N = F_3^1 = \{f_1, f_2\}$, and the shaded nodes identify subtrees that crossover could use to replace *N*.

3.3 Overloaded Types

In `CGP2.1` type information has been integrated, but in a different capacity than in STGP. First, types could be used to automatically generate type-based constraints (in place of/in addition to *Tspecs*). However, types can also be used to define overloaded functions - those generating different types (and thus spanning different constraints) based on types of their actual arguments. This is similar to having context-based constraints. For additional information and examples refer to *CGP2.1 User's Manual* (Janikow, on-line).

Example5 Assume domain types are $length, number$. Then, $multiply(a, b)$ could be overloaded to generate *number*s when applied to *number*s, and *length* when arguments are mixed. Two *length* arguments may not be allowed - unless $length^2$ is also defined as a possible type.

3.4 Heuristic (weighted) Constraints

`CGP2.1` also allows heuristic constraints, which are similar to *weak* constraints of many optimization problems. This capability allows for preferences rather than absolute constraints to be processed. For example, in the domain of boolean functions, one may desire that boolean or be *twice* as likely to apply to and as to itself. Constraints of this form could be used to identify, for example, DNF (disjunctive normal) forms as those preferable for evolution, while still allowing other forms to coexist without penalizing them. For more information and examples refer to *CGP2.1 User's Manual* (Janikow, on-line).

3.5 Language for File Constraints

`CGP2.1` allows the constraints to be listed in a file, using specially crafted language. Syntax and details can be found in (Janikow, on-line). Here we present two examples.

Example6 The following file section defines *T*- and *Fspecs* for the inverse kinematics problem:

```
FTSPEC
F_(*)=           #not required since it's empty
F_(*)[*]=        #not required since it's empty
F_(sin)[0]=add   #prevent sin(_+_)
F_ROOT=asin      #prevent asin() from Root
#must specify some TSpecs
T_(*)[*]=*       #allow all TSpecs
T_ROOT=*         #allow all func/term for Root
ENDSECTION
```

Example7 The following file section defines types and overloads some functions for the same problem.

```
TYPE
TYPELIST = float integer angle    #list valid types
(add)(float float)=float     #float + float = float
```

```
(add)(integer float)=float    #integer + float = float
(add)(float integer)=float    #float + integer = float
(add)(integer integer)=integer
                              #integer + integer = integer
(add)(angle angle)=angle   #angle + angle = float
(asin)(float)=angle
                  #asin can take float and integer only
(asin)(integer)=angle
(sin)(angle)=float
            #sin can take anything producing angle
(1)=integer                   #terminal types
(PI)=angle                    #PI is an angle, not a float
(x y)=float
ROOT=angle                    #Root's return type
ENDSECTION
```

4. Illustrative Experiment with Multiplexer

In this section, we follow a practical example intended to illustrate how *T*- and *Fspecs* can be used to control CGP's search space even though there are no type constraints - boolean is the unique type and thus closure is trivially satisfied. This will also illustrate how the search space affects GP evolution.

We use the widely studied 11-multiplexer problem: Koza (1992). 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, $a_0 \ldots a_2$ and $d_0 \ldots d_7$, respectively. For example, 110 address (the boolean formula $a_2 a_1 \overline{a_0}$), causes d_6 ($110_2 = 6_{10}$) be passed to the output. 11-multiplexer implements a boolean function, which can be expressed in DNF (disjunctive normal form) as:

$$a_2 a_1 a_0 d_7 \vee a_2 a_1 \overline{a_0} d_6 \vee a_2 \overline{a_1} a_0 d_5 \vee a_2 \overline{a_1}\,\overline{a_0} d_4 \vee$$
$$\overline{a_2} a_1 a_0 d_3 \vee \overline{a_2} a_1 \overline{a_0} d_2 \vee \overline{a_2}\,\overline{a_1} a_0 d_1 \vee \overline{a_2}\,\overline{a_1}\,\overline{a_0} d_0$$

Koza (1992) has proposed to use functions $F_I = \{if, or, and, not\}$ and terminals $F_{II} = \{a_0..a_2, d_0..d_7\}$. In this case, GP evolves trees which are labeled with the above primitive elements, each element having the standard interpretation. Evaluation assigns raw fitness value based on the number of the possible 2048 input combinations which produce the correct classification (True or False).

The function set is obviously complete, thus satisfying *sufficiency*. However, the set is also redundant – a number of 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 redundancy in the representation space.

However, as the results indicate, some combinations of subspaces will perform a much better search than others. This will be further elaborated.

For each experiment, we repeat 10 independent runs, with a population of 2000, 0.85/0.1/0.05 probabilities for crossover/selection/mutation, and otherwise the default *lil-gp* parameters. We report the same learning curves while running for 100 iterations.

To make the presentation more systematic, we assume that *Tspecs* stay constant, and all constraints are expressed with *Fspecs*. Because there is a single type, the generic *Tspecs* do not impose any constraints:

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

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

1. Experiment E_0: Unconstrained 11-multiplexer with *CGP lil-gp*. There are no constraints (all *Fspecs* are empty), thus the runs are the same as in standard GP.

2. Experiment E_1: using a 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 functions. This is be accomplished 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 solution 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. Results support this claim.

3. Experiment E_2: DNF. We attempt to generate DNF 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 only. This can be expressed (one of possible options) with the following *Fspecs*:

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

4. 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:

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

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

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

Previous experience with other evolutionary algorithms using DNF representation suggest that DNF is “the right” representation: Janikow (1993). 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.

5. Experiment E_4: using {*if*} only. Here we observe that the function set $F_I = \{if\}$ is completely sufficient for the task of learning the 11-multiplexer. This experiment, and its further enhancements, will have the best learning characteristics. Restricting trees to use this function only can be accomplished with the following *Fspecs*:

   $F^{Root} = F^*_{if} = \{or, and, not\}$, $F_* = \varnothing$,

   $F^*_{or} = F^*_{and} = F^*_{not} =$ irrelevant

6. Experiment E_5: E_4 with problem-specific knowledge. Now, suppose that, in addition to observing that $\{if\}$ is a sufficient 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 conditional 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 CGP theory it was possible to specify the stronger constraint for this function set, because that theory was based on domain sets rather than functions). This can be expressed with the following *Fspecs*:

   $F^{Root} = \{or, and, not, a_0, a_1, a_2\}$, $F_* = \varnothing$,

   $F^1_{if} = \{or, and, not, d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$,

   $F^2_{if} = F^3_{if} = \{or, and, not, a_0, a_1, a_2\}$,

   $F^*_{or} = F^*_{and} = F^*_{not} =$ irrelevant

7. 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:

   $F^{Root} = \{or, and, not, a_0, a_1, a_2\}$, $F_* = \varnothing$,

   $F^1_{if} = \{if, or, and, not, d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$,

   $F^2_{if} = F^3_{if} = \{or, and, not, a_0, a_1, a_2\}$,

   $F^*_{or} = F^*_{and} = F^*_{not} =$ irrelevant

The results are very interesting, some even striking. Forcing evolution with $\{and, not\}$ functions (E_1) has a disastrous effect, even though in theory it dramatically reduces the number of redundant solutions being explored. This indicates that simply reducing search spaces may not be beneficial. Moreover, 11-multiplexer expressions using only $\{and, not\}$ are generally more complex.

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 to another evolutionary program designed for machine learning. GIL: Janikow (1993) 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 20% of the 2048 cases were available for evaluation, GIL was evolving 99% correct solutions with respect to all the 2048 cases after exploring about 50,000 individuals. Our DNF-constrained CGP evolved less than 90%-perfect solutions after exploring 200,000 individuals, while seeing all 100% of the possible cases. Even 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 used only blind crossover/mutation, controlled by 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.

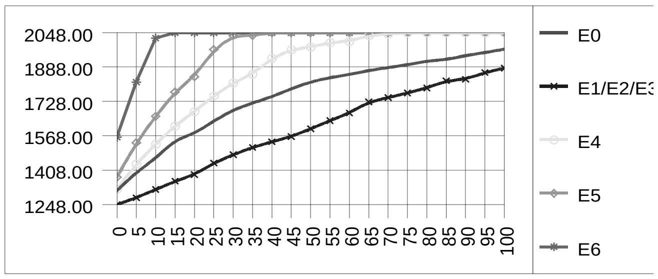

Figure 3 Comparison of the quality of the best-of-population tree. Because of similar results, E1, E2, and E3 are averaged.

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. Results for E_4 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: Janikow (1996).

This result indicates that it is indeed important to provide both 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 “unnecessary” subspace was pruned away.

Finally, providing additional heuristics about the desired solutions, and thus pruning away other undesired solutions, leads to even better speed ups (E_5 and E_6 in Figure 3). This further supports our observation that providing such information is advantageous not only to generating solutions with some specific characteristics but to speeding up evolution as well. Unfortunately, this can usually be done only by a careful redesign of the algorithm/representation/operators, or the function set in GP. In CGP, no changes are needed.

5. Summary

This paper describes `CGP2.1`, which provides a means for pruning constraint-identified subspaces from being explored in GP search. The constraints are specified 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; when studied systematically, it may also give insights about pruning redundant subspaces from any state-space search.

`CGP2.1` implements constraints based on explicit function/terminal inclusion and exclusion, mostly applicable to single-type problems (as in section 5). This capability allows for specifying constraints not based on types but on other criteria. `CGP2.1` also supports explicit types and constraints based on those types (similarly to STGP). The same explicit types can also be used for overloading functions, which capability corresponds to context-specific constraints. Moreover, additional heuristics can be expressed with weighted constraints (preferences). This last capability is currently being extended to evolve such preferences simultaneously with evolving problem solutions. Hard-coded experiments (not reported) indicate that representation heavily favoring $\{if\}$, experimentally determined to be superior in section 5, automatically evolves even if no initial constraints are given to the same problem. This extension, if indeed successful, will have far-reaching implications for GP.

Bibliography

[1] Leonard Bolc and 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] John Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.

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

[6] Cezary Z. Janikow. *"A Methodology for Processing Problem Constraints in Genetic Programming". Computers and Mathematics with Applications*, Vol. 32, No. 8, pp. 97-113, 1996.

[7] Cezary Z. Janikow. `CGP2.1. http://www.cs.umsl.edu/~janikow/cgp-lilgp`.

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

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

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

[11] Zbigniew Michalewicz & Cezary Z. Janikow. *"GENOCOP: A Genetic Algorithm for Numerical Optimization Problems with Constraints", Communications of the ACM.* Vol 39, No. 12, VE (on-line, `http://www.acm.org/cacm/extension/michalew.pdf`), 1996.

[12] David Montana. *"Strongly Typed Genetic Programming". Evolutionary Computation,* Vol. 3, No. 2, 1995.

[13] Janusz Wnek, J. Sarma, A. Wahab & R.S. Michalski. *"Comparing Learning Paradigms via Diagramatic Visualization".* In M. Emrich, Z. Ras & M. Zemankowa (eds.) *Methodologies for Intelligent Systems 5.* North Holland, 1990.

[14] Douglas Zongker & Bill Punch. *"lil-gp 1.0 User's Manual".* `zongker@isl.cps.msu.edu`.