ADAPTABLE CONSTRAINED GENETIC PROGRAMMING: EXTENSIONS AND APPLICATIONS

Final Report

NASA Faculty Fellowship Program – 2004

Johnson Space Center

Prepared By:	Cezary Z. Janikow, Ph. D.
Academic Rank:	Associate Professor
University and Department:	University of Missouri - St. Louis Department of Mathematics and Computer Science St. Louis, Missouri 63121
NASA/JSC	
Directorate:	Engineering
Division:	Automation, Robotics and Simulation
Branch:	Intelligent Systems
JSC Colleague:	Dennis Lawler
Date Submitted:	July 22, 2004
Contract Number:	NAG 9-1526 and NNJ04JF93A

ABSTRACT

An evolutionary algorithm applies evolution-based principles to problem solving. To solve a problem, the user defines the space of potential solutions, the representation space. Sample solutions are encoded in a chromosome-like structure. The algorithm maintains a population of such samples, which undergo simulated evolution by means of mutation, crossover, and survival of the fittest principles. Genetic Programming (GP) uses tree-like chromosomes, providing very rich representation suitable for many problems of interest. GP has been successfully applied to a number of practical problems such as learning Boolean functions and designing hardware circuits.

To apply GP to a problem, the user needs to define the actual representation space, by defining the atomic functions and terminals labeling the actual trees. The sufficiency principle requires that the label set be sufficient to build the desired solution trees. The closure principle allows the labels to mix in any arity-consistent manner. To satisfy both principles, the user is often forced to provide a large label set, with *ad h*oc interpretations or penalties to deal with undesired local contexts. This unfortunately enlarges the actual representation space, and thus usually slows down the search. In the past few years, three different methodologies have been proposed to allow the user to alleviate the closure principle by providing means to define, and to process, constraints on mixing the labels in the trees. Last summer we proposed a new methodology to further alleviate the problem by discovering local heuristics for building quality solution trees. A pilot system was implemented last summer and tested throughout the year.

This summer we have implemented a new revision, and produced a User's Manual so that the pilot system can be made available to other practitioners and researchers. We have also designed, and partly implemented, a larger system capable of dealing with much more powerful heuristics.

INTRODUCTION

Genetic programming (GP), proposed by Koza [1], is an evolutionary algorithm, and thus it solves a problem by utilizing a population of solutions evolving under limited resources. The solutions, called chromosomes, are evaluated by a problem-specific user-defined evaluation method. They compete for survival based on this evaluation, and they undergo simulated evolution by means of simulated crossover and mutation operators.

GP differs from other evolutionary methods by using trees to represent potential problem solutions. Trees provide a rich representation that is sufficient to represent computer programs, analytical functions, and variable length structures, even hardware circuits. The user defines the representation space by defining the set of functions and terminals labeling the nodes of the trees (for the internal and the external nodes, respectively). One of the foremost principles is that of *sufficiency* [1], which states that the function and terminal sets must be sufficient to solve the problem. The reasoning is obvious: every solution will be in the form of a tree, labeled only with the user-defined elements. In the absence of specific knowledge and heuristics, sufficiency will usually force the user to artificially enlarge the sets to avoid missing some important elements. This unfortunately dramatically increases the search space.

Even disregarding sufficiency, GP practitioners still face another problem. Consider a 3-argument *if* function (corresponding to the *if-else* conditional statement in any programming language). This function should have a test argument, and then two action arguments. But GP has no way of defining or processing this knowledge. To allow GP to operate nevertheless, Koza has proposed the principle of *closure* [1], which requires very elaborate semantic interpretations to ensure the validity of any arity-consistent label in any context. Structure-preserving crossover was introduced as the first attempt to handle such specific local constraints [1].

Structure-preserving crossover wasn't a generic method. In the nineties, three independent generic methodologies were developed to allow problem-independent constraints on tree construction. Montana proposed STGP [5], which used types to control the way functions and terminals can label local subtrees. For example, the function *if* can be required to use Boolean-producing subtrees on its first argument.

We proposed CGP, developed at NASA/JSC during summer research, which originally required the user to explicitly specify allowed and/or disallowed local tree structures [2]. These local constraints could be based on types, but also on some problem specific heuristics. In a follow-up version, we also added explicit type-processing capabilities, with polymorphic functions. For example, the + function could be overloaded so that it produces an integer from integers but it produces an angle from angles.

Finally, those interested in program induction following specific syntax structure have used similar ideas to propose CFG-based GP [6]. However, those systems still require the user to enter all possible constraints that can be processed within the methodology. What happens if the user is not aware of any, not aware of the best constraints to solve a particular problem, or aware of the constraints but not of some rule-of-a-thumb heuristics? To deal with this case, last summer we introduced Adaptable Constrained

Genetic Programming v1.1 (ACGP1.1), which is a methodology (and a pilot implementation) to automatically adapt the user-specified constraints and heuristics to improve GP's performance. This summer, we have improved that implementation (ACGP1.1.1), and provided a User's Manual, so that the system can be distributed to interested practitioners and researchers.

ACGP1.1 discovers limited local heuristics and only on labels. This summer, we have designed a new methodology, ACGP2.1, which is capable of discovering much richer heuristics on labels, types, and combinations of these. We have implemented the system, capable of discovering seven different heuristics. In the next few months, we plan to build tools to analyze the heuristics, and then to use them to improve GP's searching capabilities.

ACGP HEURISTICS

ACGP is a methodology to discover useful heuristics on GP solution trees. Such heuristics, if available, have been shown to greatly enhance GP problem solving capabilities [2]. ACGP discovers the heuristics by observing the distribution of labels (and types if applicable) in those better-off solutions, assuming that those solutions are better because they were generated with better distribution of local heuristics on average. ACGP1.1.1 deals with limited heuristics on labels, while the newer ACGP2.1 handles heuristics on both labels and types.

ACGP1.1.1 and First-Order Label-Based Heuristics

Last year we have developed ACGP1.1, which processes heuristics on labels only, while limited to parent-child relationship only. That is, ACGP1.1 doesn't process type information, and it cannot discover any heuristics taking a node's siblings into account. We call this kind of limited heuristics the *first-order* heuristics, as illustrated in Figure 1 center. *Zero-order* heuristics use one node at a time, while *second-order* heuristics take all siblings into account.

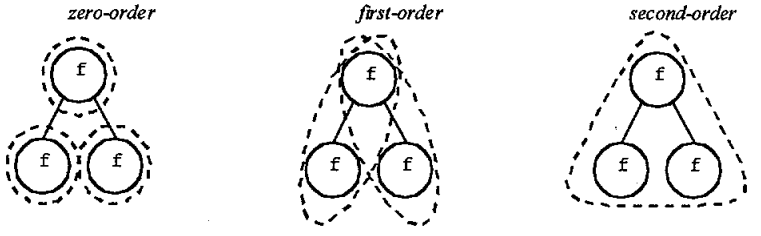

Figure 1: The three different levels of heuristics. Note that zero-order heuristics are meaningless if the only node information is the node's label.

ACGP1.1.1 does not process type information, and thus it only uses a function/terminal label in a node. This is illustrated in Figure 2 (left). Apparently, this limited data provides no information for zero-order heuristics. ACGP1.1.1 does not employ any second-order heuristics either, as it was used as a proof of concept. Even the limited first-order

heuristics have been demonstrated to both allow the user to discover very useful knowledge and to improve problem-solving capabilities [3][4].

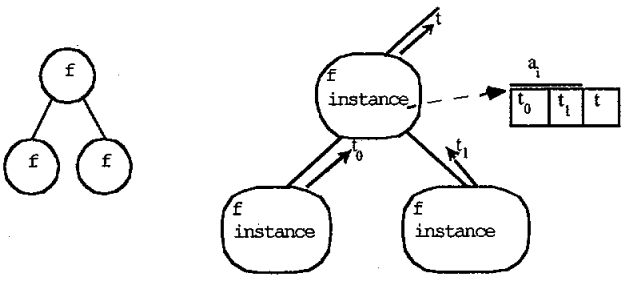

Figure 2: Information available in a tree node: left in ACGP1.1.1 (label only), right in ACGP2.1 (label, type generated, and the polymorphic function instance used).

ACGP2.1

ACGP2.1 uses both labels and types, with polymorphic functions. Therefore, the information retained in a tree node is much richer, as illustrated in Figure 2 (right): each node retains the label (function for an internal node and terminal for an external node) and the specific polymorphic instance (for functions only – terminals are not overloaded). For example, in Figure 2 right, the top node retains the information that the function used in the node is **f**, and that the function generates type **t**, using its overloaded instance which requires types $\mathbf{t_0}$ and $\mathbf{t_1}$ on the two arguments, left to right respectively.

The richness of this information allows useful zero-order heuristics. For example, the same node in Figure 2, if expressed disproportionally in the better solutions, would suggest that **f** should use this polymorphic instance whenever it generates type **t**. ACGP2.1 uses a number of heuristics of all three kinds: zero, first, and second-order. Moreover, it uses heuristics on labels only (for typeless applications), on types only, and on combinations of these. Example heuristics are illustrated in Figures 3-6.

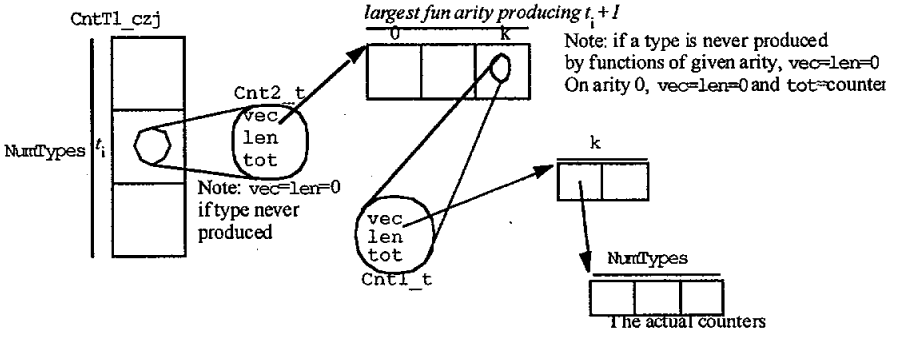

Figure 3: First-Order Type-Based Heuristics CntT1.

Figure 3 illustrates a very specific first-order heuristic on types only. This particular heuristic is capable of discovering, for example, that if a given type needs to be generated

(t_i), what arity function would be most beneficial to generate it, and for that arity, what should be the types of all of the arguments.

Figure 4 illustrates a specific first-order heuristic on labels only. This heuristic is capable of discovering what should be the labels of all the children, independently, of a function such as f_i. This heuristic alone is in fact equivalent to all the capabilities available in ACGP1.1.1.

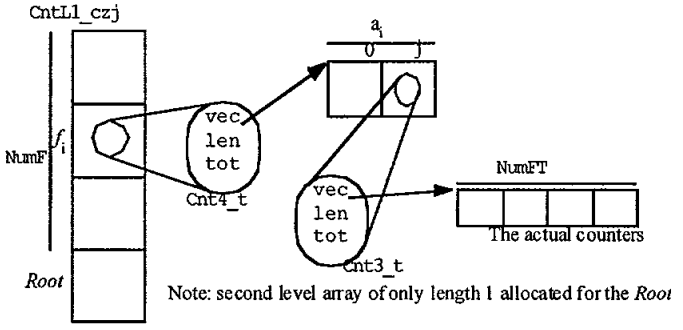

Figure 4: First-Order Label-Based Heuristics CntL1.

When we combine labels and types, even one node provides some meaningful information, resulting in combined zero-order heuristics. One such heuristic is illustrated in Figure 5. It is capable of discovering that if a given type needs to be generated from a node (such as t_i), what functions/terminals are most likely to generate it successfully.

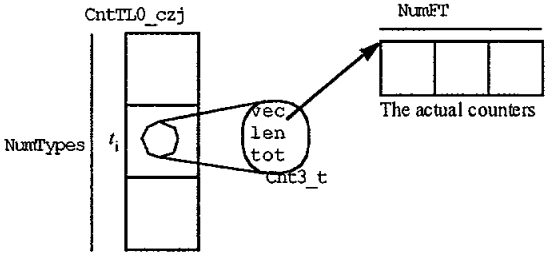

Figure 5: Zero-order Combined Heuristics CntTL0.

Figure 6 illustrates another combined zero-order heuristic. This one is capable of discovering that if a given node needs to be labeled with a function such as f_i, which polymorphic type it should use to be most successful (terminals are not polymorphic).

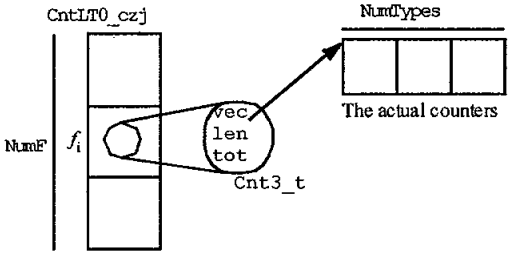

Figure 6: Zero-order Combined Heuristics CntLT0.

Currently, ACGP2.1 discovers seven such heuristics, printing them into 28 files (using different scenarios). In the near future, tools to analyze, and to use the heuristics in improving the search, are needed.

REFERENCES

[1] Koza, J. R. 1994, *Genetic Programming. On the Programming of Computers by Means of Natural Selection*, Massachusetts Institute of Technology.

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

[3] Janikow, Cezary Z. "*ACGP: Adaptable Constrained Genetic Programming*". *Proceedings of GPTP04* TBP.

[4] Janikow, Cezary Z. "*Adapting Representation in Genetic Programming*". *Proceedings of GECCO 2004.*

[5] Montana, D. J 1995, "*Strongly Typed Genetic Programming*", *Evolutionary Computation*, Vol. 3, No. 2.

[6] Whigham, P.A.. "*Grammatically-based genetic programming*". In J. P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, Tahoe City, California, USA, 9 1995.