Adapting Representation in Genetic Programming

Cezary Z Janikow

Department of Math and Computer Science
University of Missouri–St. Louis
St Louis, MO 63121 USA
cjanikow@ola.cs.umsl.edu

Abstract. Genetic Programming uses trees to represent chromosomes. The user defines the representation space by defining the set of functions and terminals to label the nodes in the trees. The sufficiency principle requires that the set be sufficient to label the desired solution trees. To satisfy this principle, the user is often forced to provide a large set, which unfortunately also enlarges the representation space and thus, the search space. Structure-preserving crossover, STGP, CGP, and CFG-based GP, give the user the power to reduce the space by specifying rules for valid tree construction. However, often the user may not be aware of the best representation space, including heuristics, to solve a particular problem. In this paper, we present a methodology, which extracts and utilizes local heuristics aiming to improve search efficiency. The methodology uses a specific technique for extracting the heuristics, based on tracing fisrt order (parent-child) distributions of functions and terminals. We illustrate those distributions, and then we present a number of experimental results. ...

1 Introduction

Genetic programming (GP), proposed by Koza [2], 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 computer hardware [2]. The user defines the representation space by defining the set of functions and terminals labelling the nodes of the trees. One of the foremost principles is that of sufficiency [1][2], 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, labelled only with the user-defined elements. 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 if the user is aware of the functions and terminals needed in a solution, he/she may not be aware of the best sub-set to solve a particular problem. Moreover, even if such a sub-set is identified, questions about the specific distribution of the elements of the subset may arise. One question is whether all functions and terminals should be equally available or if there should be some heuristic distribution. For example, a terminal t may be required but never as an argument to function f1, and maybe just rarely as an argument to f2. All of the above are obvious reasons for designing:

- methodologies for processing such heuristics,
- methodologies for automatically extracting those heuristics.

Methodologies for processing user heuristics have been proposed over the last few years: structure-preserving crossover [2], STGP [6], CGP [3], and CFG-based GP [8].

This paper presents a methodology for extracting such heuristics, called Adapt-able Constrained GP (ACGP). It is based on CGP, which allows for processing syntax, semantics, and heuristic constraints in GP [3]. In Section 2, we briefly describe CGP, paying special attention to its role in GP problem solving as a methodology for processing constraints and heuristics. In Section 3, we introduce the ACGP method-ology for extracting heuristics, and then present the specific technique that was implemented for the methodology. In Section 4, we define the problem we will use to illustrate the technique, illustrate the distribution of functions/terminals during evolution, and present some interesting results. Finally, in concluding Section 5, we elaborate on future work needed to extend the technique and the methodology.

2 Constraining GP Trees with CGP

Even in early GP applications, it became apparent that functions and terminals should not be allowed to mix in an arbitrary way. For example, a 3-argument if function should use, on its condition argument, a subtree that computes a Boolean and not temperature or angle. Because of the difficulties in enforcing these constraints, Koza has proposed the principle of closure [2], which requires very elaborate semantic interpretations to ensure the validity of any subtree in any context. Structure-preserving crossover was introduced as the first attempt to handle such constraints [2] (the primary initial intention was to preserve structural constraints imposed by automatic modules ADFs).

Structure-preserving crossover wasn't a generic method. In the nineties, three in-dependent generic methodologies were developed to allow problem-independent constraints on the tree construction. Montana proposed STGP [6], which used types to control the way functions and terminals can label local tree structures. For exam-ple, if the function if requires Boolean as its first argument, only Boolean-producing functions and terminals would be allowed to label the root of that subtree. Janikow proposed CGP, which originally required the user

to explicitly specify allowed and/or disallowed local tree structures [3]. These local constraints could be based on types, but also on some problem specific heuristics. In v2.1, CGP also added type-processing capabilities, with function overloading mechanisms [4]. For example, if a subtree needs to produce an integer, and we have the function + (add) overloaded so that it produces integers only if both arguments are integers, then only this specific instance of add would be allowed to label the root of that subtree. Finally, those interested more directly in program induction following specific syntax structure have used similar ideas to propose CFG-based GP [8].

CGP relies on closing the search space to the subspace satisfying the desired constraints. The constraints are local distribution constraints on labelling the tree - only parent-child relationships can be efficiently processed [3] (the processing was shown to impose only constant overhead for mutation and one more tree traversal for cross-over).

CGP v1 allows processing only parent-one-child contexts. This context constraint is independent of the position of the subtree in the tree, and of the other labels beyond this context.

Types and function overloading in v2 allows this context to be extended to the other siblings. For example, v2 allows to require that whenever a node is labelled with if, its leftmost child can only be labelled with Boolean terminals (and possibly not all Boolean terminals if there exist some heuristics), or with functions which can be instantiated to produce Boolean (these possible labelling elements form so called mutation sets in CGP). As mentioned in the conclusions, the parent-child context can be further extended in the future.

CGP has one additional unique feature. It allows a particular local context to be weighted, to reflect some detailed heuristics. For example, it allows the user to de-clare that the function if, even though it can use either f1 or f2 for its condition child, it should use f1 more likely. This feature is utilized in ACGP to express and process the heuristics.

Previous experiments with CGP have demonstrated that proper constraints can indeed greatly enhance the evolution, and thus improve problem-solving capabilities. However, in many applications, the user may not be aware of those proper constraints. For example, as illustrated with the 11-multiplexer problem, improper constraints can actually reduce GP's search capabilities while proper constraints can greatly speed up evolution [3]. This paper presents a new methodology, which automatically updates the constraints, or heuristics, to enhance the search characteristics with respect to some user-defined objectives (tree quality and size at present). In what follows, we describe the methodology and a specific technique implementing it, and then present some experimental results.

3 ACGP and the Local Distribution Technique

ACGP is a methodology to automatically modify the heuristic weights on typed mutation sets in CGP. The basic idea is that there are some heuristics on the distribution of labels in the chromosomes both at the local level (parent-child)

and at a more global level. These ideas are somehow similar to those applied in Bayesian Optimization Network [7], but used in the context of GP and functions/terminals and not binary alleles.

We have already investigated two ACGP techniques that allow such modifications. One technique observes the utility of specific local contexts when applied in mutation and crossover, and based on the utility (parent-offspring fitness relation-ship) it increases or decreases the weights for the used constraints. A very simple implementation of this technique was shown to increase GP problem solving capabilities (mutation was more problematic due to its bucket-brigade problem) [5].

In here, we investigate a similarly simple technique. It observes the distribution of functions and terminals in all/best trees (and thus the surviving distribution of all/best parent-child contexts). Note that we are using distribution to refer to the local context. Examples of such distributions are presented in Section 4. This idea is somehow similar to that used for CFG-based GP as recently reported in [8].

ACGP basic flowchart is illustrated in Fig. 1. ACGP works in iterations - iteration is a number of generations ending with extracting the distribution. The distribution information is collected and used to modify the actual mutation set weights (the heuristics). The modification can be gradual (slope on) or complete replacement (slope off). Then, the run continues, with the same population or with a randomly regrown (regrow on) population. The regrowing option seems beneficial with longer iterations, where likely some material gets lost before being accounted for in the distributions, and thus needs to be reintroduced by regrowing the population (as will be shown in Section 4). Note that the newly regrown population is generated based on new (updated) heuristics and thus may be vastly different from the first initial population - see Section 4 for illustrations.

ACGP can also work with simultaneous independent multiple populations (pop), to improve its distribution statistics - see Figure 1.4. ACGP can in fact correlate the populations by exchanging selected chromosomes - however, we have not tested such settings yet.Moreover, at present all independent populations contribute to and use the same single set of heuristics -we have not experimented with maintaining separate heuristics, which likely would result in solving the problem in different subspaces by different populations.

Each population is ordered based on a 2-key sorting, which compares sizes (ascending) if two fitness values are relatively similar, and otherwise compares fitness (descending). The more relaxed the definition of relative similarity, the more importance is placed on sizes.

Subsequently, the best use percent of the ordered chromosomes are selected into a common pool (from all populations). This pool of chromosomes is used to compute distribution statistics (or only use percent of the pool if all=0). The distribution is a 2-dim matrix counting the frequency of parent-child appearances. Table 1 illustrates some extracted context distribution. Assume the function f1 has 2 arguments (as shown), and there are 2 functions and two terminals in the user set (f1, f2, t1, t2). This function (f1) appears 100 times in the selected set

of trees (total for each row is 100). The cell f1(arg1)[f1] = 20 says that in 20 of the 100 cases the first argument subtree is labelled with f1. The 0 entry in the last cell indicates that the terminal t2 never labels the second subtree of f1 in the selected set.

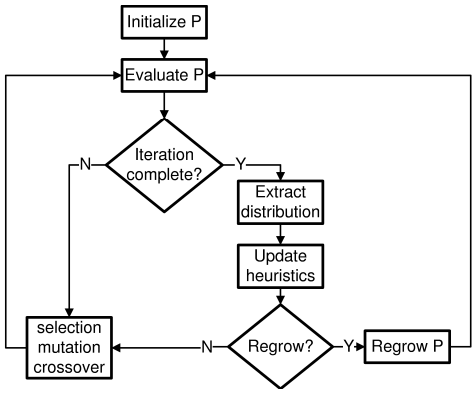


Fig. 1. The flowchart for the ACGP algorithm

Table 1 illustrates some extracted context distribution. Assume the function f1 has 2 arguments (as shown), and there are 2 functions and two terminals in the user set (f1, f2, t1, t2). This function (f1) appears 100 times in the selected set of trees (total for each row is 100). The cell f1(arg1)[f1] = 20 says that in 20 of the 100 cases the first argument subtree is labelled with f1. The 0 entry in the last cell indicates that the terminal t2 never labels the second subtree of f1 in the selected set.

Table 1. Examples of extracted distributions (partial matrix)

	f1	f2	t1	t2
Function f1 arg1	20	40	10	30
Function f1 arg2	10	10	80	0

4 Illustrative Experimental Results

To illustrate the concepts, we traced the local distributions in the population, measured fitness gains in subsequent iterations, and also attempted to visualize the extracted heuristics both in terms of their specific values and in terms of dynamic changes in subsequent iterations.

All reported experiments used 1000 trees per population, the standard mutation, crossover, and reproduction operators at the rate of 0.05, 0.85, and 0.1, and for the sake of sorting, trees with fitness values differing by no more than 2% of the fitness range values in the population were considered the same on fitness (and thus ordered ascending by size).

4.1 Illustrative Problem: 11-multiplexer

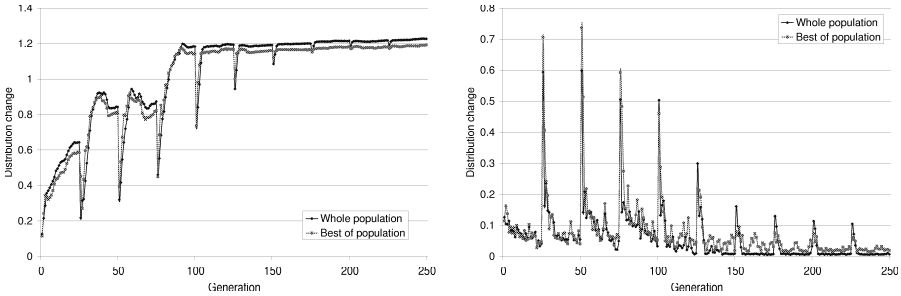
To illustrate the behavior of ACGP, we selected the well-known 11-multiplexer problem [2]. This problem is not only well known and studied, but we also know from [3] which specific constraints improve the search efficiency. Our objective was to attempt to discover some of the same constraints automatically, and to observe how they change the search properties over multiple ACGP iterations.

The 11-multiplexer problem is to discover the Boolean function that passes the correct data bit (out of eight d0d7) when fed 3 addresses (a0...a2). There are 2048 possible combinations. Koza [2] has proposed a set of 4 atomic functions to solve the problem: 3-argument if/else, 2-argument and, or, and 1-argument not, in addition to the data and address bits. This set is not only sufficient but is also redundant. In [3] it was shown that operating under a sufficient set such as not with and degrades the performance, while operating with only if (sufficient by itself) and possibly not improves the performance. Moreover, it was shown that the performance is further enhanced when we restrict the if's condition argument to choose only addresses, straight or negated, while restricting the two action arguments to select only data or recursive if [3]. This information is beneficial as we can compare ACGP-discovered heuristics with these previously identified and tested — as we will see, ACGP discovers virtually the same heuristics.

4.2 Change in Distribution of Local Heuristics

Here we traced the change in distribution of functions and terminals in the populations without utilizing ACGP, just to visualize specific characteristics and behavior of the distribution. In the future, we plan to use this information for further automation of ACGP. The distribution change is measured in terms of the local contexts, as explained in the previous section, and with respect to a reference distribution (either the distribution in the initial or in the previous generation). The distribution change is defined as the sum of squared differences between the two populations on individual frequencies (normalized cells in Table 1). This change can be measured for individual functions, function-arguments, and even function-argument-values. How-ever, unless indicated otherwise, the illustrations presented in the paper use all functions/terminals for the change measure. This subsection shows averages of five independent single-population runs.

Fig. 2a illustrates the change in the distribution of the local heuristics in the whole population, and also in just the best 20% of the population, with a) the initial random population as the reference. As seen, during the initial generations there is a rapid change in the distribution, as tress with "bad" contexts become



(a) Reference = initial population (b) Reference = previous population

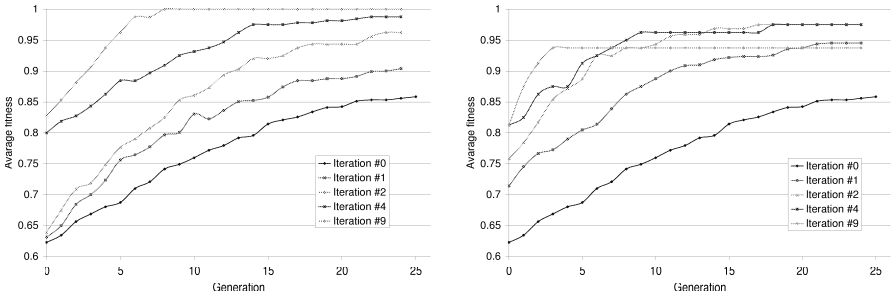
Fig. 2. Function/terminal distribution in the whole population and the best 20%

extinct. Two observations are crucial to ACGP and its future extensions. First, the change saturates, which will become important if ACGP were to automatically trace the change to decide when to modify its heuristics (at present, the user decides that). Second, the best trees have distributions very similar to those in the best population.

One may suggest that even though the changes saturate, it doesn't mean that the actual changes from generation to generation subside. Fig. 2b) illustrates that in fact population-to-population changes diminish to result in the saturation see in Fig. 2a.

4.3 Change in the Speed of Evolution

The first obvious question regarding ACGP's performance is what good it does. In this section, we answer the question in terms of average fitness growth and the number of generations needed to find a solution, while in the next section we look at the extracted quantitative and qualitative heuristics.

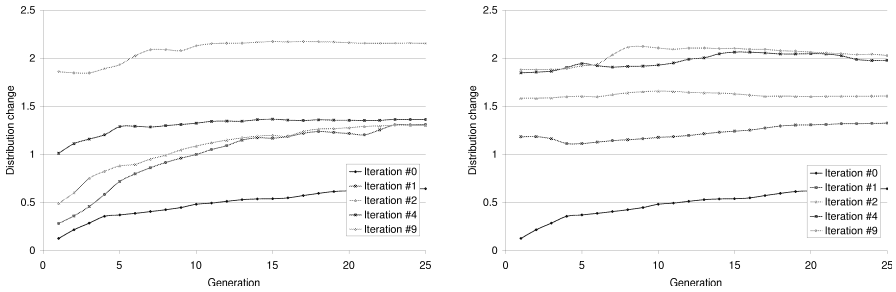


(a) Update heuristics (slope=on) (b) Replace heuristics (slope=off)

Fig. 3. Average fitness across five populations in selected iterations (with regrow)

We have conducted two experiments while running five independent CGP populations. In both experiments, we have adjusted the heuristic weights every 25 generations (one iteration), with the parameter regrow set on, for a total of 10 iterations. The first experiment used incremental weight changes (slope on), while the other immediately replaced the weights with the actual extracted distributions, and used the all option (used the distribution from all the extracted trees). Fig. 3 presents the results, shown separately for each of the 10 iterations. As seen, the average fitness grows much faster in subsequent iterations, indicating that ACGP did indeed extract helpful heuristics. Moreover, the initial fitness in the initial random population of each iteration (regrow causes each new iteration to start with a new random population) also increases. As illustrated later in Fig. 4, some populations could eventually solve the problem in the generation in subsequent iterations. Between the two, we can see that the second run (slope off) causes much faster learning but it is too greedy and indeed fails to solve the problem consistently (average saturation below 1.00 fitness). Inspection of the evolved heuristics revealed that some terminals in the evolved representation had very low weights, making it harder to consistently solve the problem in all populations even though it made it easier to "almost" solve the problem (in fact, the same experiment with all off completely dropped one terminal off the representation, making it impossible to solve the problem).

Fig. 4a illustrates the same results differently. It plots the number of generations needed to solve the problem in subsequent iterations by the best individuals in the population (not average). Here we can also see that the greedy slope-off case indeed reduces the number of required generations much faster and after the third iteration it always solves the problem on the first generation! However, as seen in Fig. 3b, not all populations were able to do so, and thus this approach was too greedy.



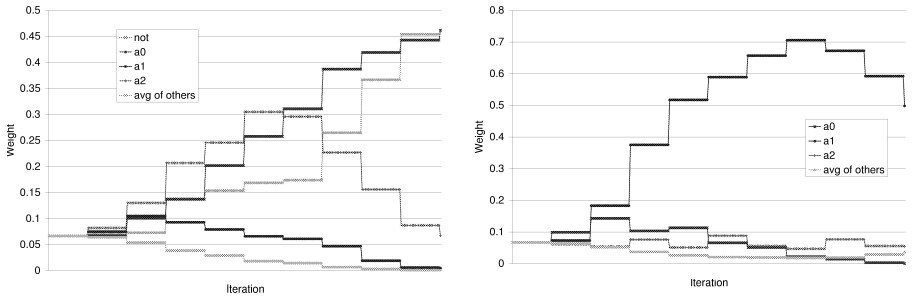
(a) Update heuristics (slope=on) (b) Replace heuristics (slope=off)

Fig. 4. Distribution changes in the whole population vs. the initial population, for each iteration separately

Finally, we also traced the same distribution change as in the previous section, for different iterations. The results are presented in Fig. 4b, which shows

distribution changes in the whole population, with the initial population of the given iteration as the reference. As seen, not all iterations contributed the same to the extracted heuristics. The first generation in fact didn't do as well as iterations 2 and 3. One may speculate that the initial iteration without any input heuristics found it too difficult to decide on the direction of change.

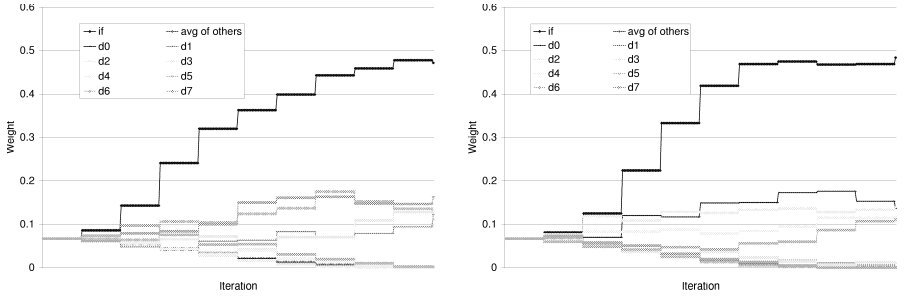
4.4 The Evolved Heuristics



(a) Heuristics on function if test argument (b) Heuristics on function *not*

Fig. 5. Evolved heuristics for if, the condition argument (direct and indirect through *not*

In this section we look at the evolved heuristics, attempting to understand them and compare against those previously identified and tested for this problem. Recall that the best results were obtained with only the if function, the three addresses and not function in the condition subtree of if, and then recursive if and the data bits in the two action subtrees [3].

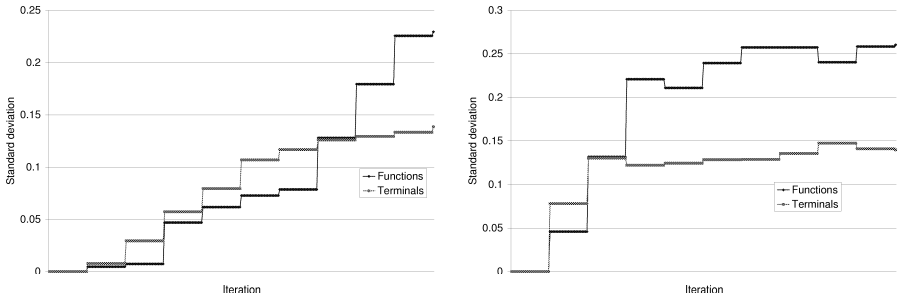


(a) Heuristics on function if first action (b) Heuristics on function if second action

Fig. 6. Evolved heuristics for if, action arguments

Fig. 5 illustrates the evolution of the heuristics on the condition part of if. All functions and terminals start equally (no prior heuristics on the first iteration). In the course of the 10 iterations, we can observe that and, or, and the data bits are indeed dropped off. We can also see a random shift, in iteration 8, between a0 and a2 (notice the corresponding increase in the required generations at this iteration in Fig. 4 left, meaning this was an accident and some work was needed to recomputed the heuristics). In fact, after generation 150, the heuristics seem more natural than at the end - one of the subsequent research topics will be to decide when to stop, based on the observed distributions. One notable observation in Fig. 5 left is that a0 is sufficiently represented, a2 is under-represented, and a1 is disallowed! Now, if this were the end of the story, we should not be finding perfect solutions so quickly using this evolved representation. Fig. 5 right reveals the answer to this puzzle. According to Fig. 5a, , not is also highly allowed in the if's condition part. According to Fig. 5b, not supports the under-represented a2, and then strongly supports the missing a1.

Fig. 6 illustrates the total heuristics on the action parts of if. Recall that the action part in the best scenario should allow only data bits, and then recursive if. Moreover, if should have higher probability to allow deeper trees. We can see that this is indeed what has evolved. The function if spikes, and the data bits remain relatively stable - however because the remaining functions/terminals drop off, in fact the data bits are extracted. In the future, we plan to extend the heuristics not only to types but also to different levels - in this case we would observe that the if weight diminishes with node depth.



(a) Update heuristics (slope on) (b) Replace heuristics (slope off)

Fig. 7. Standard deviation on the function/terminal heuristics for function if test argument

Finally, we observe the changes in the heuristics of the if's condition argument by tracing the standard deviation in its distribution of functions and terminals, separately for both. The results are illustrated in Fig. 7. Of course the deviations start very low on the first generation of the first iteration (driven by the initially equal heuristics) - note the log scale. At subsequent iterations, the weights become more diverge as the heuristics are learned, and they also

change less (relative to the iteration), indicating again that heuristics extraction has saturated.

4.5 Influence of Population Size, Sampling Rate, and Iteration Length

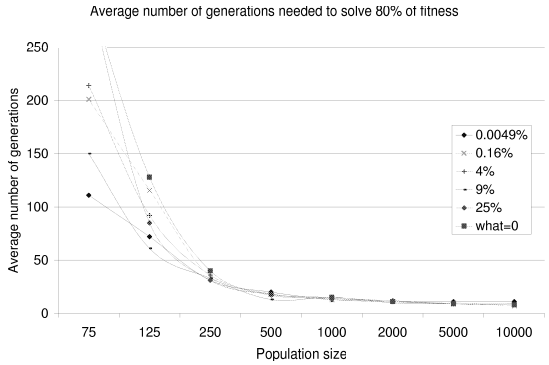


Fig. 8. The number of generations needed to solve for 80% – sampling rate for distributions is the effective rate.

We have only reported here experiments for Iteration=25, population=1000, and sampling of 4% (20% of 5 populations, then 20% of that set). Fig. 8 presents a cummulative result for iteration=1, various population sizes, and various effective sampling rates. As seen, ACGP works quite well with short iterations, and in fact beats GP especially for smaller populations and smaller sampling rates. It seems that ACGP allows smaller populations to solve the same problem. For more of such experiments, refer to [5].

5 Conclusions

This paper presents the ACGP methodology for automatic extraction of heuristic constraints in genetic programming. It is based on the CGP methodology, which allows processing such constraints and heuristics. The ACGP algorithm here implements a technique based on distribution of local first-order (parent-child) contexts in the population. As illustrated, ACGP is able to extract such heuristics, which not only improve search capabilities but also have meaningful interpretation as com-pared to previously determined best heuristics for the 11-multiplexer problem.

The paper also illustrates the changes in the distributions in the population, and identifies specific characteristics that can be used to further automate

ACGP. This is what we plan to investigate in the future, along with other techniques for extracting the heuristics, such as co-evolution of heuristics and solutions. We also plan to extend the technique to types and overloaded functions as available in CGP v2.1. This will allow processing a child with its parent in the context of its sibling.

Other questions to be explored include:

- Extending the technique to the CGP v2 technology, which allows overloaded functions, and this extending the heuristics to the context of siblings.
- Linking population size with ACGP performance and problem complexity.
- Scalability of ACGP.
- Varying the effect of distribution and the heuristics at deeper tree levels.
- Exchanging chromosomes or separating heuristics between populations.
- Clustering techniques to explore useful high-order heuristics (more levels deep). This is similar to ADFs, except that ACGP would learn clusters of deep heuristics rather than abstract functions.
- The resulting trade-off between added capabilities and additional complexity when using deeper heuristics (CGP guarantees its low overhead only for first-order constraints/heuristics).
- Other techniques for the heuristics, such as co-evolution between the heuristics and the solutions.

References

1. Banzhaf, W, et al.: Genetic Programming, and Introduction. Morgan Kaufmann (1998)
2. Koza, J. R.: Genetic Programming. On the Programming of Computers by Means of Natural Selection. Massachusetts Institute of Technology (1994)
3. Janikow, C.Z.: A Methodology for Processing Problem Constraints in Genetic Programming. Computers and Mathematics with Applications, Vol. 32, No. 8 (1996) 97–113
4. Janikow, C.Z. and Deshpande, R.A.: Evolving Representation in Genetic Programming. Proceedings of ANNIE'03 (2003) 45–50
5. Janikow, C.Z.: ACGP: Adaptable Constrained Geneting Programming. Proceedings of GPTP (2004) TBP
6. Montana, D. J.: Strongly Typed Genetic Programming. Evolutionary Computation, Vol. 3, No. 2 (1995)
7. Pelikan, M. and Goldberg, M.: BOA: The Bayesian Optimization Algorithm. Proceedings of GECCO'99 (1999) 525–532
8. Shan, Y., McKay, R. I., Abbass, H. A., and Essam, D.: Program Evolution with Explicit Learning: a New Framework for Program Automatic Synthesis. Technical Report. School of Com-puter Science, Univ. College, Univ. of New South Wales. Submitted Feb. 2003.
9. Whigham, P. A.: Grammatically-based Genetic Programming. In: J. P. Rosca (ed.): Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications (1995) 33–41