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, the user often 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 at improving search efficiency. The methodology uses a specific technique for extracting the heuristics, based on tracing first-order (parent-child) distributions of functions and terminals. We illustrate these distributions, and then we present a number of experimental results. . . .

1 Introduction

Genetic programming (GP), proposed by Koza [2], solves a problem by utilizing a population of solutions evolving under limited resources. The solutions (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 [1][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* [2], which states that the function and terminal sets must be sufficient to solve the problem. The reason is obvious: every solution will be in the form of a tree, labelled only with the user-defined elements. Sufficiency usually forces 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 domain, he/she may not be aware of the best subset to solve a particular problem. Moreover, even if such a subset 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 in every context, or whether 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 [9].

This paper presents a methodology for extracting such heuristics, called *Adaptable Constrained GP* (ACGP). It is based on the technology of CGP, which allows for efficient processing syntax, semantics, and heuristic constraints in GP [3]. In Sec. 2, we briefly describe the CGP technology. In Sec. 3, we introduce the ACGP methodology for extracting heuristics, and then present the specific distribution-based technique that was implemented for the methodology. In Sec. 4, we define the problem we will use to illustrate the technique, trace the first-order distribution of functions/terminals during evolution, and present some results. Finally, in concluding Sec. 5, we elaborate on future work needed to extend the technique and the methodology.

2 The CGP Technology

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/else* 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 usually 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 independent 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 example, 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. 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 (*e.g.*, [9]).

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) — the processing was shown to impose only constant overhead for mutation and one more tree traversal for crossover [3].

CGP v1 allowed 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 (even the siblings)[1]. CGP v2 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 declare that the function *if*, even though it can use either *f1* or *f2* for its condition child, it should use *f1* more frequently. This particular efficient technology 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[2] in CGP. The basic idea is that there are some heuristics on the distribution of labels in the trees both at the local level (parent–child) and at a more global level (currently not done). 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 their utility (parent–offspring fitness

[1] Types and function overloading in v2 allows this context to be extended to the other siblings. This provides technology to extend the current first-order distribution.

[2] CGP expresses its heuristics by so called weighted mutation sets [3].

relationships) it increases or decreases the weights for the applied heuristics. A very simple implementation of this technique was shown to increase GP problem solving capabilities. However, mutation was much more problematic due to its bucket-brigade problem [4], and thus the overall improvements were marginal.

In here, we investigate a similarly simple technique, one that 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 Sec. 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 population undergoing the standard reproduction, or with a randomly regrown (*regrow on*) population. The regrowing option was found 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 reported in [5], regrowing is destructive for shorter iterations). Note that the newly regrown population is generated based on new (or updated) heuristics and thus may be vastly different from the first initial population — see Sec. 4 for illustrations.

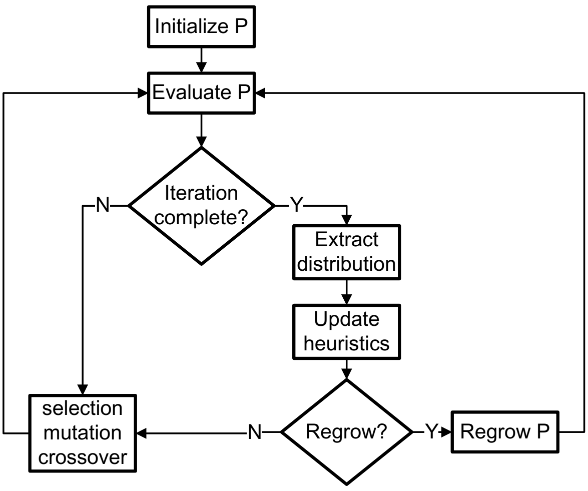


Fig. 1. The flowchart for the ACGP algorithm

ACGP can also work with simultaneous independent multiple populations, to improve its distribution statistics. 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 *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 fitness similarity, the more importance is placed on sizes.

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

Subsequently, the best *use* percent of the ordered chromosomes are selected into a common pool (from all populations) and resorted again. This pool of chromosomes is used to compute distribution statistics (from all or from *use* percent of the top chromosomes). The distribution is a *2-dim* matrix counting the frequency of parent-child appearances. Table 1 illustrates some extracted context distributions. 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 root of the first subtree is also 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 chromosome set.

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 quantitatively and qualitatively, as compared to those previously identified to be beneficial.

All reported experiments used 1000 trees per population, five populations, 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). Unless otherwise noted, all experiments report the average of the best of five populations, and they executed with *iteration* = 25 generations and *regrow on*.

4.1 Illustrative Problem: 11-multiplexer

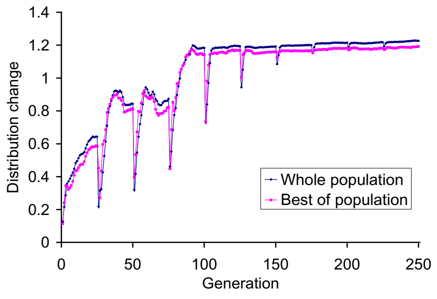
To illustrate the behavior of ACGP, we selected the well-known 11-multiplexer problem [2]. This problem is not only well known and studied, but we also know from [3] which specific constraints improve the search efficiency. Our objective was to attempt to discover some of the same constraints automatically, 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 *d0... d7*) when controlled by three addresses (*a0... a2*). There are 2048 possible combinations. Koza [2] has proposed a set of four atomic functions to solve the problem: 3-argument *if/else*, 2-argument *and* and *or*, and 1-argument *not*, in addition to the data and address bits. This set is not only sufficient but also redundant. In [3] we have shown that operating under a sufficient set such as {*not, and*} degrades the performance, while operating with only *if* (sufficient by itself) and possibly *not* improves the performance. Moreover, we have 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 prior 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.

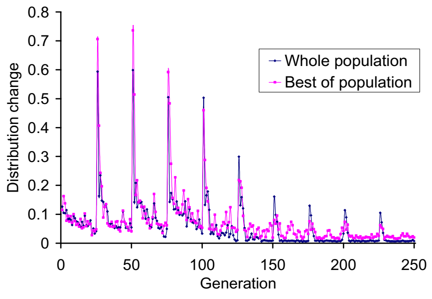
4.2 Change in Distribution of Local Heuristics

Here we traced the change in distribution of functions and terminals in the populations, just to visualize specific characteristics and behavior of the distribution. The distribution change is measured in terms of the local contexts, as explained in the previous section, and with respect to a reference distribution (the distribution in either 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 (*e.g.*, normalized cells in Table 1). This change can be measured for individual functions, function–arguments, and even function–argument–values. However, unless indicated otherwise, the illustrations presented in the paper use all functions/terminals for the change measure.

Fig. 2 illustrates the change in the distribution of the local heuristics in the whole population, and also in just the best 4% of the population, when executed on the 11-multiplexer problem with *iteration*=25 and *slope on* (graduate change in heuristics). Fig. 2a uses initial population as the reference. As can be seen, the distribution grows with generations, but then diminishes upon regrowing the population at the end of each iteration. However, overall each new random population becomes more distant from the original initial population — an illustration of the fact the the heuristics change causes different distribution of labels in each regrown population. Moreover, the changes eventually saturate — as we will see shortly, this is because the heuristics have been extracted. Fig. 2b uses the previous population as the reference. As can be seen, each regrow dramatically changes the population (spikes) — however, the changes eventually



(a) Reference = initial population



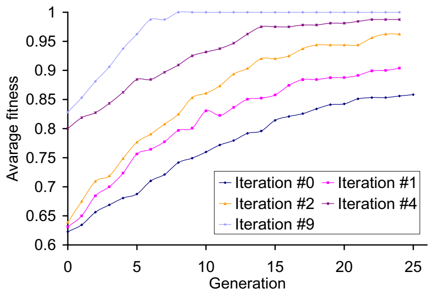
(b) Reference = previous population

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

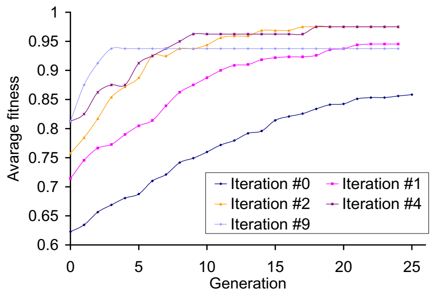
diminish, indicating (as see Fig. 3) that even the randomly regrown populations contain high quality chromosomes.

4.3 Change in the Speed of Evolution

The obvious question is what is the effect of the new heuristics on the learning curves. In this section, we answer the question in terms of average fitness growth, while in the next section we look at the extracted quantitative and qualitative heuristics.



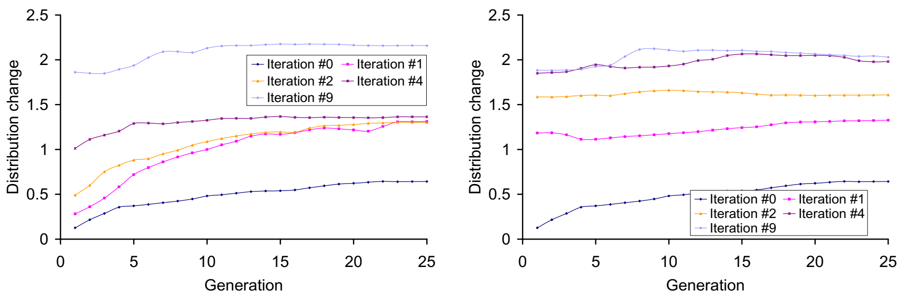
(a) Update heuristics (*slope on*)



(b) Replace heuristics (*slope off*)

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

We have conducted two experiments, with *iteration*=25 and total of 250 generations (10 iterations). In one experiment, we adjusted the heuristics (*slope on*) while in the other we replaced the heuristics with those from the distribution (*slope off*). 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, which can be helpful in subsequent runs (in Fig. 8 we show that ACGP can also improve on the first run vs. standard GP). 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. Between the two, we can see that the second case (*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 smaller sampling completely dropped one terminal from the representation, making it impossible to solve the problem).



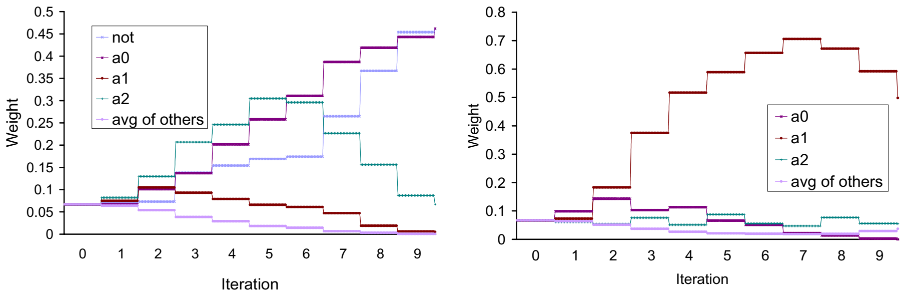
(a) Update heuristics (*slope=on*) (b) Replace heuristics (*slope=off*)

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 individual iterations. The results are presented in Fig. 4, which shows distribution changes in the whole population, with the initial population as the reference. They support the conclusions from Fig. 3 — *slope off* causes much faster changes in the distributions, but as seen before this can be too greedy. It will be important in the future to be able to predict the trade off between the pace of extraction of heuristics and and loss of representative power.

4.4 The Evolved Heuristics

In this section, we look at the evolved heuristics, attempting to qualify 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 (directly or through *not*) in the condition subtree of *if*, and then recursive *if* and the data bits in the two action subtrees [3].



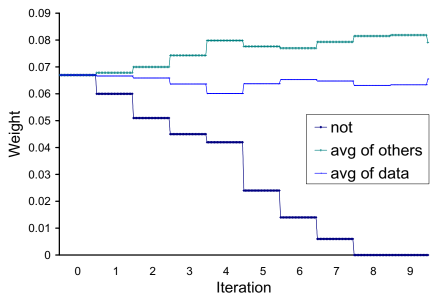
(a) Heuristics on function *if* test argument

(b) Heuristics on function *not*

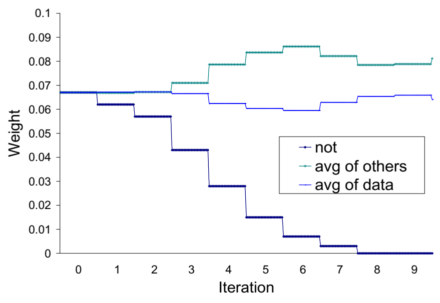
Fig. 5. Evolved heuristics for *if*, the condition argument (direct and indirect through *not*

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 ten iterations, we can observe that *and*, *or*, and the data bits are indeed dropped off (average shown). We can also see that the best heuristics indeed evolved around iteration six–seven (in fact the learning curves dropped off slightly after this point). This illustrates the need for other means to determine termination of the process. Fig. 5a illustrates the direct heuristics on *if* condition argument. As seen, *not* is highly present (to allow indirect addresses, see Fig. 5b). Among the address bits, *a0* is highly present, *a2* is marginally present, and *a1* is virtually absent. However, as seen in Fig. 5b, *a1* is very highly supported through indirect *not*, while *a2* has additional support as well.

Fig. 6 illustrates the total heuristics on the two 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 hopefully observe that the *if*'s weight diminishes with node depth.

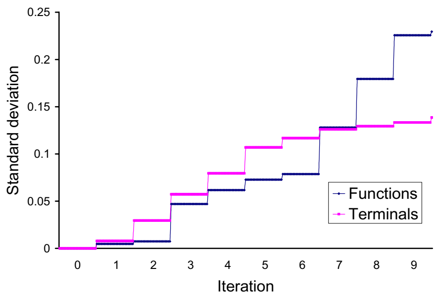


(a) Heuristics on function *if* first action

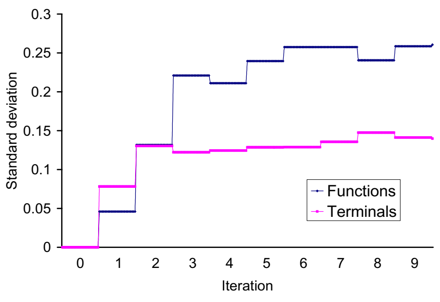


(b) Heuristics on function *if* second action

Fig. 6. Evolved heuristics for *if*, action arguments



(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 iteration (driven by the initially equal heuristics). At subsequent iterations, the weights become more diverse as the heuristics are learned, and they also change less (relative to the iteration), indicating again that heuristics extraction has saturated. Between the two, again slow updates (a) in heuristics led to slower changes, while drastic replacement of heuristics (b) led to faster changes and faster saturation. One more interesting observation is the sudden spike in terminals distribution for the update case (a), which coincides with the previously identified iteration where we observed the sudden shift in the heuristics (Fig. 5a).

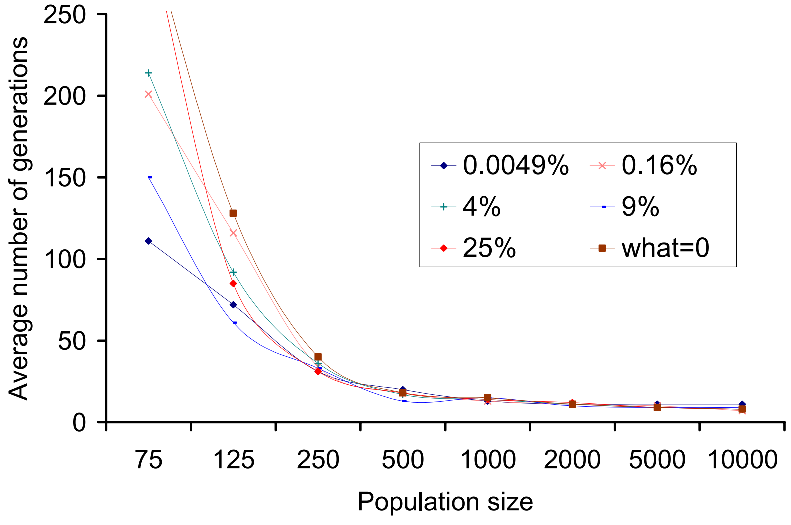


Fig. 8. The number of generations needed to solve for 80% — sampling rates for distributions are the effective rates (what=0 indicated a plain GP run)

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

So far, we have reported experiments with long iterations (25 generations), population of 1000 trees (per population), and using *regrow* at the beginning of each new iteration. The results indicate that ACGP can extract meaningful heuristics, which helps in solving the problem fast and better in subsequent iterations. This is fine if the objective is to extract the heuristics or solve the problem with unlimited time. However, one may wish to make those improvements when solving the problem for the first time as well, while also extracting the heuristics. In this section, we relate population size needed to solve the problem, while attempting to learn and use the heuristics on the first execution of the system. More extensive results are reported in [5]. Fig. 8 presents accumulative result for *iteration*=1, various population sizes, and various effective sampling rates (effective rate refers to the percentage of trees eventually used for distribution with respect to all the trees in all the populations). 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 faster. However, much more studies are needed here.

5 Conclusions

This paper presents the ACGP methodology for automatic extraction of heuristic constraints in genetic programming. It is based on the CGP technology, which allows efficient processing of such constraints and heuristics. The ACGP algorithm presented 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 compared to previously determined best heuristics for the same problem.

The paper also illustrates the changes in the distributions in the population, and raises a number of questions for the future.

- Extending the technique to the CGP v2 technology, which allows overloaded functions, and thus extending the heuristics to the context of siblings.
- Linking population size with ACGP performance and problem complexity.
- Determining 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 technology guarantees its low overhead only for the one–level 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