Chapter 1

# ACGP: ADAPTABLE CONSTRAINED GENETIC PROGRAMMING

Cezary Z. Janikow
*Department of Math and CS*
*UMSL*
janikow@umsl.edu

**Abstract** GP requires that all functions/terminals (tree labels) be given apriori. In the absence of specific information about the solution, the user is often forced to provide a large set, thus enlarging the search space — often resulting in reducing the search efficiency. Moreover, based on heuristics, syntactic constraints, or data typing, a given subtree may be undesired or invalid in a given context. Typed GP methods give users the power to specify some rules for valid tree construction, and thus to prune the otherwise unconstrained representation in which GP operates. However, in general, the user may not be aware of the best representation space to solve a particular problem. Moreover, some information may be in the form of weak heuristics. In this work, we present a methodology, which automatically adapts the representation for solving a particular problem, by extracting and utilizing such heuristics. Even though many specific techniques can be implemented in the methodology, in this paper we utilize information on local first–order (parent–child) distributions of the functions and terminals. The heuristics are extracted from the population by observing their distribution in "better" individuals. The methodology is illustrated and validated using a number of experiments with the 11-multiplexer. Moreover, some preliminary empirical results linking population size and the sampling rate are also given.

**Keywords:** Genetic Programming, representation, learning, adaptation, heuristics

## 1. Introduction

Genetic Programming (GP), proposed by Koza [4], 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 fitness, and they undergo simulated evolution by means of crossover and mutation operators.

GP differs from other evolutionary methods by using different representation, usually 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 [4, 1]. 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* [4], 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 ensure that no important elements are missing. 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 subset to solve a subproblem (that is, used locally in the tree). Moreover, even if such subsets are identified, questions about the specific distribution of the elements of the subsets may arise — should all applicable functions and terminals have the same uniform probability in a /inxxprobability distribution given context? For example, a terminal $t$ may be required, but never as an argument to function $f_1$, and maybe just rarely as an argument to function $f_2$. All of the above are obvious reasons for designing methodologies for:

- processing such *constraints* and *heuristics*,
- automatically extracting those constraints and heuristics.

Methodologies for processing user constraints (that is, strong heuristics) have been proposed over the last few years: structure–preserving crossover [4], type–based STGP [5]), type, label, and heuristic–based CGP [2], and syntax–based CFG–GP [8].

This paper presents a methodology, called *Adaptable Constrained GP* (ACGP), for extracting such heuristics. It is based on CGP, which allows for processing syntax, semantic, and heuristic–based constraints in GP [2]). In Section 2, we briefly describe CGP, paying special attention to its role in GP problem solving as a technology for processing constraints and heuristics. In Section 3, we introduce the ACGP methodology for extracting heuristics, and then present the specific technique, distribution statistics, that was implemented for the methodology. In Section 4,

we define the 11-multiplexer problem that we use to validate the technique, illustrate the distribution of functions/terminals during evolution, and present some selected results in terms of fitness curves and extracted heuristics. Moreover, we also present some interesting empirical results linking population size, ACGP, and sampling rate for the distribution. Finally, in concluding Section 5, we elaborate on current limitations and future work needed to extend the technique and the methodology.

## 2. 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 $if$ function should use, on its condition argument, a subtree that computes a Boolean and not a temperature or angle. Because of the difficulties in enforcing these constraints, Koza has proposed the principle of *closure* [4]), which allows any arity–consistent labelling, often accomplished through elaborate semantic interpretations. The working environment for such a GP system is illustrated in Figure 1.1a — initialization, and then mutation and crossover choose from the complete set of functions and terminals, with uniform distribution.

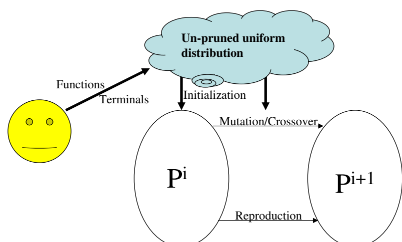


*Figure 1.1a.* Working environment for a standard GP.

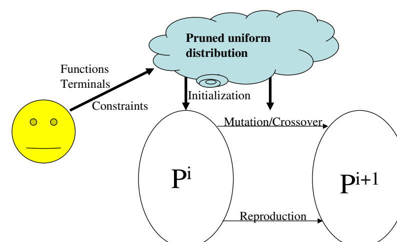


*Figure 1.1b.* Working environment for a typed GP.

Structure–preserving crossover was introduced as the first attempt to handle some strong constraints [4] (the initial primary initial intention was to preserve structural constraints imposed by automatic modules — ADFs). In the nineties, three independent generic methodologies were developed to allow problem–independent strong constraints on tree construction. Montana proposed STGP [5], which uses types to control the way functions and terminals can label local tree structures. For example, if the function $if$ requires a 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 labels in different contexts [2]. These local constraints could be based on types, but also on some problem specific semantics. In v2.1, CGP also added explicit type–based constraints, along with polymorphic functions. Finally, those interested more directly in program induction following specific syntax structure, have used similar ideas in CFG–based GP [8].

CGP relies on closing the search space to the subspace satisfying the desired constraints. That is, only trees valid with respect to the constraints are ever processed. This is helped by the guarantee that all operators produce constraints–valid offspring from constraints–valid parents [2]. The allowed constraints, type–based, or explicitly provided, are only those that can be expressed in terms of *first–order constraints* (that is, constraints expressed locally between a parent and one of its children). These constraints are processed with only minimal overhead (constant for mutations, one additional traversal per crossover parent) [2].

The working environment for a typed–based system such as the ones mentioned above is illustrated in Figure 1.1b — the representation space is locally pruned; however, the remaining elements are still subject to the same uniform application distribution.

CGP has one additional unique feature. It allows constraints to be weighted, in effect changing hard constraints into soft heuristics. For example, it allows the user to declare that some function $f$, even though it can use either $f_1$ or $f_2$ for its child, it should use $f_1$ more likely. Accordingly, the CGP working application environment becomes that of Figure 1.2a — with the final distribution of functions/terminals/subtrees for initialization, mutation, and crossover becoming non-uniform. This efficient technology is utilized in ACGP to express, process, and update the heuristics during evolution.

Previous experiments with CGP have demonstrated that proper constraints/heuristics 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 or heuristics. For example, as illustrated with the 11-multiplexer problem, improper constraints can actually reduce GPs search capabilities, while proper constraints can increase them greatly [2]. ACGP is a new methodology allowing automatic updates of the weighted constraints, or heuristics, to enhance the search characteristics with respect to some user–defined objectives (currently tree quality and size).

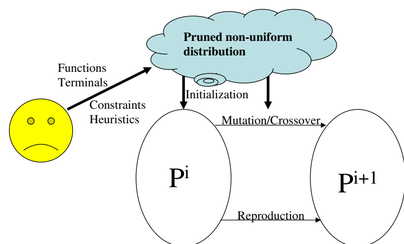


*Figure 1.2a.* Working environment for CGP.

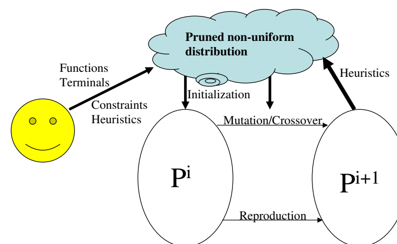


*Figure 1.2b.* Working environment for ACGP.

## 3. ACGP Methodology and the Local Distribution Technique

CGP preprocesses its input constraints into weighted mutation sets: the *mutation set* for a function $f$ is the set of functions and terminals that can label the children of $f$ (separately for all children). CGP v2.1 uses more elaborate mechanisms to process types and polymorphic functions. However, because the current ACGP methodology has not been extended to utilize those features, in what follows we will not be concerned with types and polymorphic functions (just plain constraints and heuristics).

ACGP v1 is a methodology to automatically modify the weights on typed mutation sets in CGP v1, thus to modify the heuristics during the evolution. Its working environment is presented in Figure 1.2b — the user many still provide initial constraints and heuristics, but these will be modified during the run. Of course, the obvious question is what technique to follow to do so. We have already investigated two ACGP techniques that allow such modifications. One technique is based on observing the fitness relationship between a parent and its offspring created by following specific heuristics. The heuristics are strengthened when the offspring improves upon the parent. A very simple implementation of this technique was shown to increase GP problem solving capabilities. However, mutation was much more problematic and not performing as well as crossover, due to the obvious bucket-brigade problem — in mutation, one offspring tree is produced by a number of mutations before being evaluated [3].

The second technique explores the distribution statistics of the first–order contexts (that is, one parent — one child) in the population. Examples of such distributions are presented in Section 4. This idea is

somehow similar to that used for CFG–based GP as recently reported [7], as well as to those applied in Bayesian Optimization Network [6], but used in the context of GP and functions/terminals and not binary alleles.

## ACGP Flowchart and Algorithm

ACGP basic flowchart is illustrated in Figure 1.3a. ACGP works in *iterations* — iteration is a number of generations ending with extracting the distribution and updating the heuristics. During a generation on which iteration does not terminate, ACGP runs just like GP (or CGP). However, when an iteration terminates, ACGP extracts the distribution information and updates the heuristics. Moreover, afterwards, the new population can be regrown from scratch (but utilizing the new heuristics) if the *regrow* option is set. 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).

The distribution information is collected from just the best samples. This information is subsequently used to modify the actual mutation set weights (the heuristics). The modification can be gradual (*slope* parameter on) or a complete replacement (*slope* off).

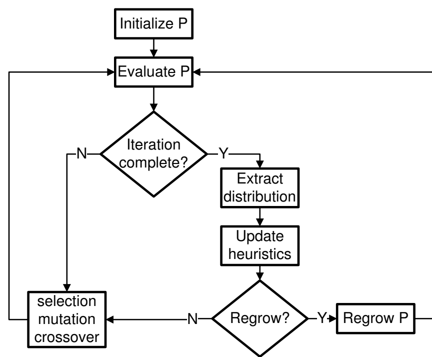


*Figure 1.3a.* ACGP basic flowchart loop.

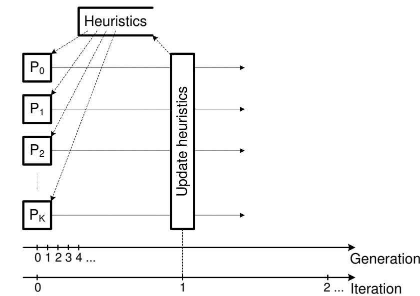


*Figure 1.3b.* ACGP1.1 iterations.

To improve the statistics, ACGP can use simultaneous multiple independent populations. However, only one set of heuristics is currently maintained, as seen in Figure 1.3b. ACGP can in fact correlate the populations by exchanging chromosomes. We have not experimented with this option, nor we did with populations maintaining separate heuristics

— which likely would result in solving the problem in different subspaces (via different constraints and heuristics) by different populations.

All trees are ordered with 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. The best trees (according to a percentage parameter) from individual populations are collected, resorted, and the final set is finally selected. This set is examined to extract the distribution and update the heuristics.

## Distribution Statistics

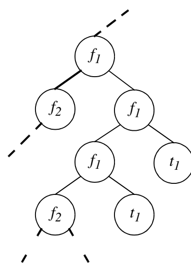


*Figure 1.4.* Sample partial GP tree.

The distribution is a 2-dim matrix counting the frequency of parent-child appearances. For example, if the tree fragment of Figure 1.4 is in the selected pool, its partial distribution matrix would be as illustrated in Table 1.1. If these were the only extracted statistics, and the *slope* was off, at the end of an iteration heuristics would be updated so that if there is a node labelled $f_1$, and its right child needs a new subtree from mutation (initialization in *regrow*) or crossover, the tree brought by crossover (or the node generated by mutation) would be 1/3 likely to be labelled with $f_1$ and 2/3 with $t_1$.

*Table 1.1.* Examples of extracted distributions

| | $f_1$ | $f_2$ | $t_1$ | $t_2$ |
|---|---|---|---|---|
| Function $f_1$ *arg1* | 1 | 2 | 0 | 0 |
| Function $f_1$ *arg2* | 1 | 0 | 2 | 0 |

### Off–line vs. On–line Environment

ACGP methodology can be used in two different settings. If our goal is to extract some knowledge about a particular problem or domain, to collect domain heuristics into a library , or to learn some heuristics for a simpler version of a problem in order to improve problem–solving capabilities when solving a more complex instance of the problem — we may run the system in *off–line* manner, meaning the heuristics are not extracted until the evolution converges. Iterations spanning over multiple generations are examples of these approaches.

On the other hand, if our goal is to solve a particular problem with the minimum effort and maximum speed, we would extract the heuristics as often as possible, possibly every generation — thus shortening iteration length to just one generation. This is the *on–line* environment.

Most of the experiments reported in Section 4 were conducted off-line, with just a few on-line results.

## 4. Illustrative Experiments

To illustrate the methodology and the distribution technique, we use the 11-multiplexer problem. Unless otherwise noted, all reported experiments used 1000 trees per population, the standard mutation, crossover, and reproduction operators at the rate of 0.10, 0.85, and 0.05, tournament (7) selection, and for the sake of sorting, trees with fitness values differing by no more than 2% of the fitness range in the population were considered the same on fitness (and thus ordered ascending by size). Moreover, unless otherwise noted, all results are averages of the best of five independent populations while executed with a single set of heuristics.

### Illustrative Problem: 11-multiplexer

To illustrate the behavior of ACGP, we selected the well–known 11-multiplexer problem first introduced to GP in [4]. This problem is not only well known and studied, but we also know from [2] which specific constraints improve the search efficiency — thus allowing us to qualitatively and quantitatively evaluate the learned here heuristics.

The 11-multiplexer problem is to discover a function that passes the correct data bit (out of eight $d_0 - d_7$) when fed three addresses ($a_0 - a_2$). There are 2048 possible combinations. Koza (1994) has proposed a set of four atomic functions to solve the problem: 3-argument $if$, 2-argument *and*, *or*, and 1-argument *not*, in addition to the data and address bits. This set is not only sufficient but also redundant. In [2] 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 $if$ condition argument to choose only addresses, straight or negated (through *not*), while restricting the two action arguments to select only data or recursive $if$ [2]. Again, this information is beneficial as we can compare ACGP–discovered heuristics with these previously identified and tested.

First, we trace a specific off–line run, observing the population distribution dynamics, the change in fitness, and the evolved heuristics. Then, we empirically test the relationship between iteration length, regrowing option, and fitness. Finally, we empirically test the relationship between population size, sampling rate, and the resulting fitness, for an on–line case.

## Off-line Experiment

In this section we experiment with *regrow* on, iteration=25 generations, 10 sequential iterations, gradual update of heuristics (*slope* on), and 4% effective rate for selecting sorted trees for distribution statistics.

**Distribution.** We trace the distribution changes separately for the entire population (average of 5 populations is shown) and the selected best samples. Figure 1.5a illustrates the distribution change in a population when compared with the initial population. As seen, the distribution difference grows rapidly (each population diverges from the initial one), but eventually saturates.

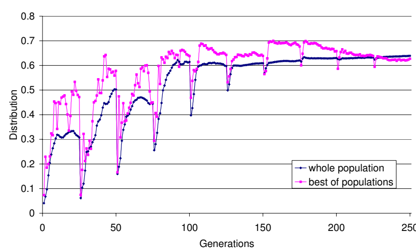


*Figure 1.5a.* Distribution of the $if$ function as a parent, with the initial population as the reference.

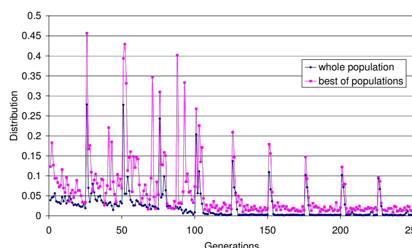


*Figure 1.5b.* Distribution of the $if$ function as a parent, with the previous generation as the reference.

Even though the distribution diverges from that of the initial population, does it converge to a single set of heuristics? The answer is

provided in Figure 1.5b — it illustrates the same distribution difference when compared to the previous population. As seen, the changes diminish over subsequent iterations, except of narrow spikes when the populations are regrown. Therefore, the heuristics do converge.

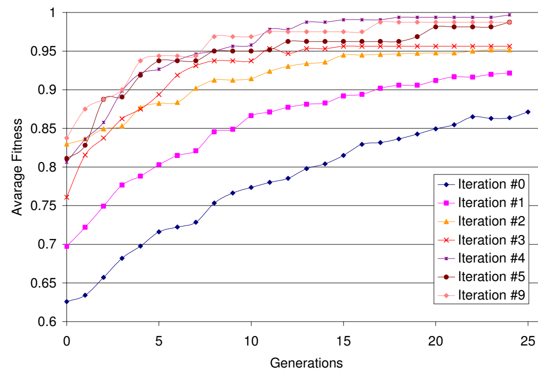


*Figure 1.6.* Fitness growth, shown separately for each iteration (*slope* on).

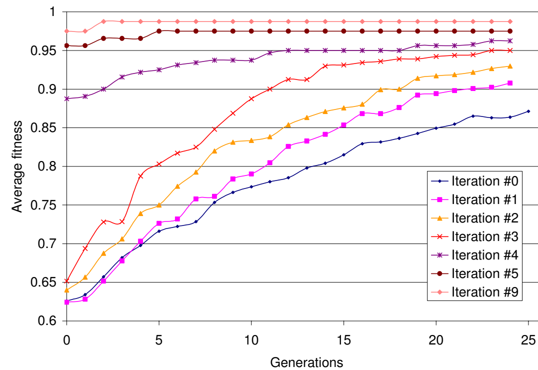


*Figure 1.7.* Fitness growth, shown separately for each iteration (*slope* off).

**Fitness.** The next two figures illustrate the resulting fitness (average of the best of each of the five populations) over sequential iterations: *slope* on (Figure 1.6), resulting in gradual changes in the heuristics, and *slope* off (Figure 1.7), resulting in a greedy instantaneous replacement of heuristics on every iteration. In both cases, subsequent iterations both start with better initially regrown populations (according to the

newly acquired heuristics) and offer faster learning curves. However, the more greedy approach (Figure 1.7) offers better initializations but also saturates below 100% - one of the five populations would consistently get stuck in a local minimum.

Altogether, we may see that off–line learning does improve subsequent runs. Thus, ACGP can learn meaningful heuristics (as also illustrated in the next section), and improve on subsequent runs. Later on we will see that ACGP can improve with on–line learning as well.

**Heuristics.** Recall that [2] has empirically determined that the best fitness curves were obtained when restricting the function set to use only $if$ and *not*, with the test argument of $if$ using only $a_0 - a_2$ straight or negated, and with the other two arguments of $if$ using recursive $if$ and the data bits only.

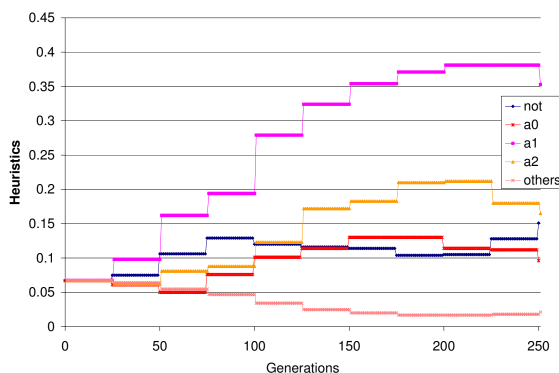


*Figure 1.8.* Direct heuristics on the test argument of $if$.

Figure 1.8 traces the evolving heuristics on the test argument over the course of the ten iterations. As illustrated, $if$ has discovered to test addresses. However, only $a_1$ is highly represented, with $a_2$ and $a_0$ lower, respectively. This does not seem like the most effective heuristic. However, the puzzle is solved when we consider that *not* is also allowed as an argument. When considering indirect heuristics (the evolved heuristics for *not*, Figure 1.9), we can see that $a_0$ and $a_2$ are supported, with reversed proportions, and with $a_1$ virtually absent - since it was already highly supported directly.

Figure 1.10 illustrates the evolved heuristics for the action arguments of $if$. As seen, recursion is highly evolved (to build deeper trees with

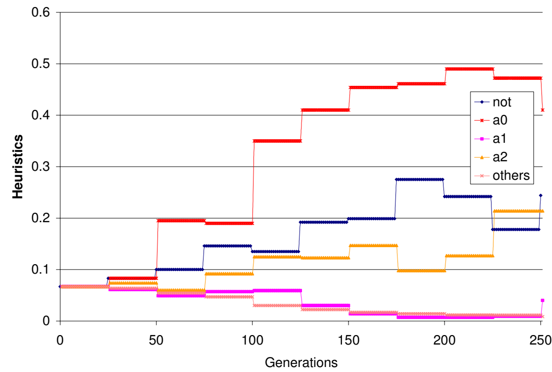


*Figure 1.9.* Indirect heuristics on the test argument of *if* (via *not*).

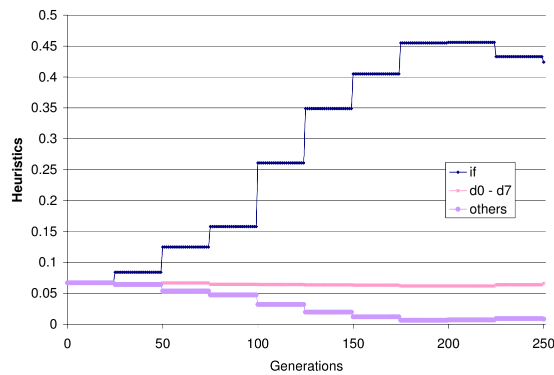


*Figure 1.10.* Combined heuristics on the two action arguments of *if*.

multiple *if*s), and all data bits are supported with the other labels all virtually disappeared.

## Varying Iteration Length and Regrow

All the results shown so far were obtained with *regrow* and off-line (iteration=25). The next question we set to assess is the impact of the iteration length and the *regrow* option on the problem solving capabilities.

We set four separate experiments, with iteration=1, 5, 10, and 25 generations, respectively. In all cases we used *slope* on and off without noticeable differences. We used both *regrow* ($what = 2$) and no *regrow* ($what = 1$), and compared against a baseline plain GP ($what = 0$).

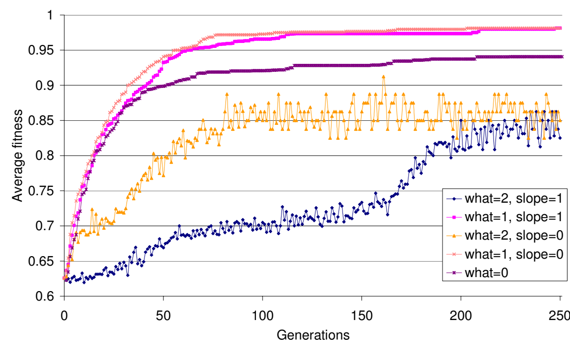


*Figure 1.11.* Fitness growth for iteration = 1 generation.

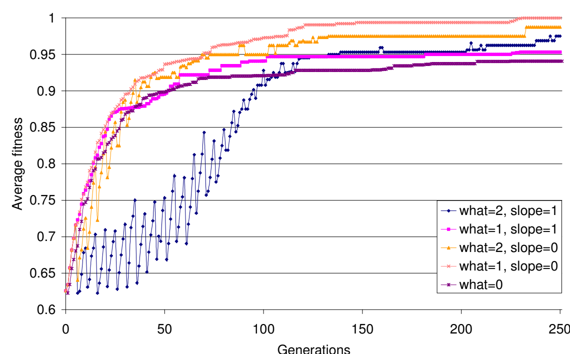


*Figure 1.12.* Fitness growth for iteration = 5 generations.

The results are shown in Figure 1.11, 1.12, 1.13 and Figure 1.14. On–line runs, as seen in the top figures, suffer from the *regrow* option, but those without *regrow* beat the standard GP. In fact, the lengthened iteration to 5 generations does provide quality solutions even with *regrow*, but in one case it takes longer to obtain those solutions.

Off-line runs, as seen in the second figure, clearly benefit from *regrow*, especially for the longer iterations. Again, this may be justified by allowing sufficient time to converge to meaningful heuristics, but with this convergence it is advantageous to restart with a new population to avoid clear saturation.

As the iteration length decreases, regrowing causes more and more harm. This is apparent - there is no time for the GP to explore the space, and the run becomes a heuristics-guided random walk.

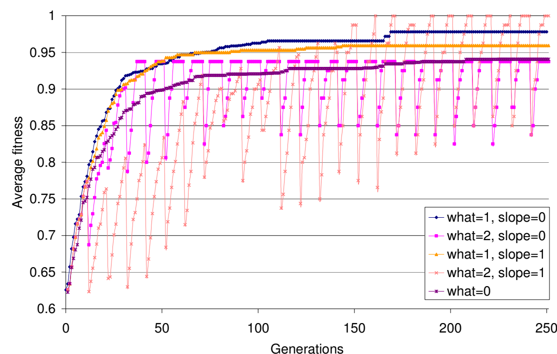


*Figure 1.13.* Fitness growth for iteration = 10 generations.

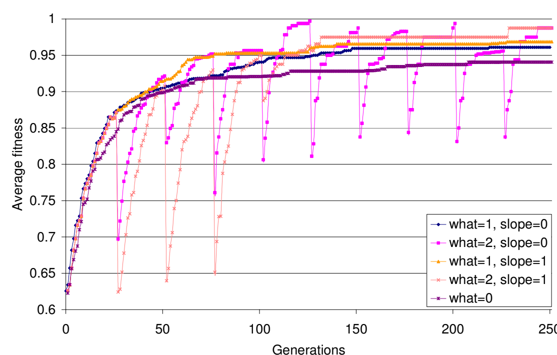


*Figure 1.14.* Fitness growth for iteration = 25 generations.

## Varying Population and Sampling Sizes

All the previous runs were conducted with the assumed population size 1000 and effective sampling rate 4% for the trees contributing to the heuristics.

In this section, we empirically study the relationship between population size, sampling rate, and the resulting fitness. All results in this section were obtained with on–line runs (iteration=1 generation).

Figure 1.15 illustrates the average fitness of the best individuals from the 5 populations, after 50 generations, as a function of the population size. The top curve is that of a plain GP. As expected, the 50 generations lead to better fitness with increasing population size, due to more trees sampled. ACGP under–performs, but this was expected — Figure 1.11 already illustrated that *regrow* in combination with iteration=1 is

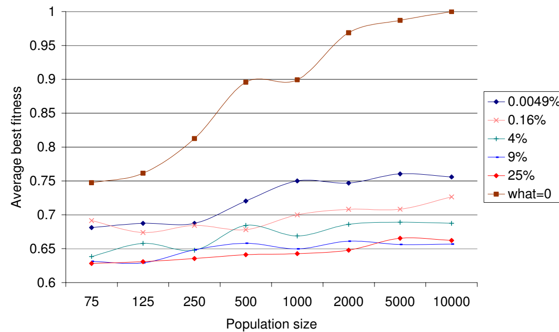


*Figure 1.15.* Average fitness after 50 generations with *regrow*.

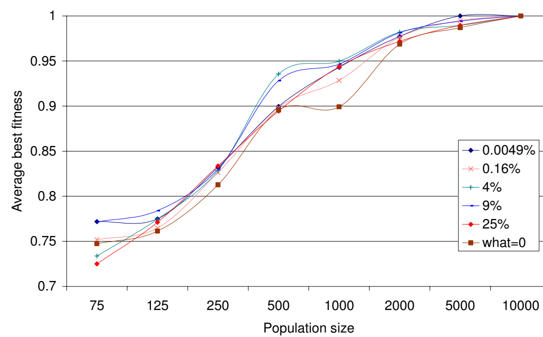


*Figure 1.16.* Average fitness after 50 generations no *regrow*.

destructive. One other observation is that decreasing effective sampling rate does improve the performance, which was observed especially for the larger populations.

Figure 1.16 presents the same fitness curves but for no *regrow*. As seen, ACGP does beat GP, especially for the lower sampling rates. Another important observation is that ACGP clearly beats GP for very low population sizes.

The same can be seen in Figure 1.17, which presents the same fitness learning for no *regrow*, but presented differently. The figure illustrates the average number of generations needed to solve the problem with at least 80% fitness. As seen, the baseline GP fails to do so for the smallest population of 75, while ACGP accomplishes that, especially with the small–sampling runs. These results, along with the others, indicate that ACGP can outperform GP especially when working with

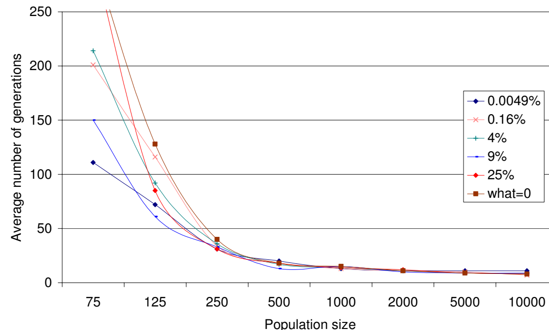


*Figure 1.17.* The number of generations needed to solve for 80% for varying population sizes.

smaller populations. One may speculate that ACGP is less dependent on population size — to be studied further in the future.

## 5. Summary

We have presented the ACGP methodology for automatic extraction of heuristics in Genetic Programming. It is based on the CGP technology, which allows processing such constraints and heuristics with minimal overhead. The ACGP algorithm implements a technique based on distribution of local first–order (parent–child) heuristics in the population.

As illustrated, ACGP is able to extract such heuristics to an advantage, and thus it performs domain learning while solving a problem at hand. Moreover, the extracted heuristics match those previously identified for this problem by empirical studies.

With the extracted heuristics, ACGP clearly outperforms a standard GP on subsequent runs (subsequent iterations) in the off–line settings, sometime solving the problem in the initial population. Moreover, with the proper setting, ACGP can also outperform GP even with the on–line settings, and it seems to be more robust with smaller population sizes.

ACGP v1 does rely exclusively on the first–order heuristics. By evaluating the resulting heuristics, one may say that the 11–multiplexer problem does possess such simple heuristics. For more complex problems, we may need to look at higher–order heuristics, such as those taking the siblings into account, or extending the dependency to lower levels. Such extensions can be accomplished by extending the distribution mechanism to compute deeper–level distributions, or by employing a Bayesian network or decision trees to the first-order heuristics.

Topics for further researched and explored include:

- Extending the technique to deeper-level heuristics.
- Using the same first–order heuristics, combined with the Bayesian network or a set of decision trees, to allow deeper-level reasoning.
- Linking population size with ACGP performance and problem complexity.
- Scalability of ACGP.
- Varying the effect of distribution and the heuristics at deeper tree levels, or taking only expressed genes into account while extracting the heuristics.
- The resulting trade–off between added capabilities and additional complexity when using deeper heuristics (CGP guarantees its low overhead only for the first–order constraints/heuristics).
- Other techniques for the heuristics, such as co–evolution between the heuristics and the solutions.
- Building and maintaining/combining libraries of heuristics with off–line processing, to be used in on–line problem solving.

## References

[1] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, dpunkt.verlag, January 1998.

[2] Cezary Z. Janikow. A methodology for processing problem constraints in genetic programming. *Computers and Mathematics with Applications*, 32(8):97–113, 1996.

[3] Cezary Z. Janikow and Rahul A Deshpande. Adaptation of representation in genetic programming. In Cihan H. Dagli, Anna L. Buczak, Joydeep Ghosh, Mark J. Embrechts, and Okan Ersoy, editors, *Smart Engineering System Design: Neural Networks, Fuzzy Logic, Evolutionary Programming, Complex Systems, and Artificial Life (ANNIE'2003)*, pages 45–50. ASME Press, 2-5 November 2003.

[4] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts, May 1994.

[5] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[6] Martin Pelikan and David Goldberg. Boa: the bayesian optimization algorithm. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 525–532, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[7] Y. Shan, R. McKay, H. Abbass, and D. Essam. Program evolution with explicit learning: a new framekwork for program automatic synthesis. Technical report, School of Computer Science, University of New Wales, 2003.

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