Evolving Problem Heuristics with On-line ACGP

Cezary Z Janikow
UMSL
St Louis, MO 63367
1-314-516-6352
janikow@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, often forcing the user to enlarge the set, thus also enlarging 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: types, syntax, and heuristics. However, in general the user may not be aware of the best representation space, including heuristics, to solve a particular problem. Recently, the ACGP methodology for extracting problem-specific heuristics, and thus for learning model of the problem domain, was introduced with preliminary off-line results. This paper overviews ACGP, pointing out its strength and limitations in the off-line mode. It then introduces a new on-line model, for learning while solving a problem, illustrated with experiments involving the multiplexer and the Santa Fe trail.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

I.2.6 [**Artificial Intelligence**]: Learning

General Terms

Design, Experimentation.

Keywords

Genetic Programming, Machine Learning, Heuristics.

1. INTRODUCTION

Evolutionary computation techniques solve 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 fitness, and they undergo simulated evolution by means of crossover and mutation operators.

Genetic Programming (GP), introduced by Koza [9] differs from other evolutionary methods by mainly using trees to represent potential solutions. Trees provide rich representation that is sufficient to represent computer programs, analytical functions, variable length structures, even computer hardware [1][9]. The user defines the representation space by defining the set of functions and terminals labeling the nodes of the trees. One of the foremost principles is that of *sufficiency* [9], which states that the function and terminal sets must be sufficient to express a solution to a problem. The reason is obvious: every solution will be in the form of a tree, labeled only with the user-defined elements. Sufficiency usually forces the user to enlarge the sets of functions and terminals, to ensure the inclusion of the necessary 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 known, another important question arises: 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()*.

Such questions are more general versions of those referred to as *design* issues. For example, McPhee with Hopper [11], and Burke [2] analyzed the effect of the root node selection on GP. Hall and Soule [4] have studied the phenomenon more extensively and have concluded that the choice of the root node has a very significant impact on the solutions generated, and that fixing the root node properly amounts to limiting the search space needed to be searched. Daida has shown that later GP generations introduce little variation into the structure of the generated trees [3], indicating that these later generations search a smaller subspace of the search space. Langdon has shown that GP typically searches only a well defined region of the potential search space [10]. Hall and Soule call these phenomena the *design* evolved by GP, arguing that this process in fact resembles *top-down design* strategy [4] – first set your choice on the most general design issues, then continue with the more specific ones. However, very little research has been devoted to such design issues beyond the root node. The reason is quite obvious – the lack of tools and methodologies to impose, observe, and analyze various designs across different tree levels.

There are two kinds of heuristics that can be available in a problem domain: *weak* and *strong*. Strong heuristics are in fact constraints, that is they impose specific rules. Weak constraints are, on the other hand, preferences. For example, a rule that prohibits *f1()* from using terminal *t* is a strong heuristic, while the rule which states that this terminal should be used half as often as other terminals is a weak heuristic.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'07, July 7–11, 2007, London, England, United Kingdom.
Copyright 2007 ACM 978-1-59593-698-1/07/0007...$5.00.

Heuristics can also be expressed at different levels, depending on the expression language. For example, one may speak of allowed labels for an argument of a function, or about two-level deep subtree structures. Of course, the more complex the language, the more detailed heuristics can be expressed.

To help process limited heuristics, some methods have been developed over the years. *Structure-preserving* crossover was introduced as the first attempt [9], with the primary initial intention to preserve structural constraints imposed by automatic modules *ADF*s. It is capable of processing very limited forms of strong constraints. In the nineties, three independent more generic methodologies have been developed to allow problem-independent heuristics on the tree construction. Montana proposed *Strongly Typed* GP (STGP) [12], which processes strong constraints based on data types. For example, if the function *f1()* requires *Boolean* as its first argument, only *Boolean*—producing functions and terminals would be allowed to label argument subtree of *f1()*. Researchers interested more directly in program induction proposed *Context-free* based GP (CFG-GP) [14], which allows processing strong context-free constraints, that is syntax rules. Janikow proposed *Constrained* GP (CGP), which allows both strong and weak heuristics [5][8].

The above technologies allow automatic processing of limited strong (plus weak in CGP) heuristics, resulting in great improvements on GP performance in domains and cases where such heuristics are necessary or known. However, what about the vast number of problems where no such heuristics are available or are unknown? This question brings two issues:

1. Evolving problem-specific heuristics as a means of improving a single GP run (on-line).
2. Evolving problem and domain-specific heuristics as a means of increasing GP's problem-solving capabilities (off-line).

Recently, a new methodology, *Adaptable* CGP (ACGP) was introduced, which allows automatic extraction of problem-specific heuristics. Such heuristics in fact represent a model of the problem, or the domain. This idea of building a model of the domain while solving the problem has been exploited in genetic algorithms with the hBOA methodology [13] or the more general Estimation Distribution Algorithms EDAs. However, the model built here is still more primitive due to the limitations of the heuristics used, as explained in section 3.

The ACGP methodology was previously demonstrated using *off-line* learning, that is learning by analyzing the trace of a particular application, and thus not immediately helping with the problem being solved [6][7]. However, when the same design was applied to *on-line* learning, that is learning to be immediately used to help with the problem being solved, the results were much worse. This paper presents a modified version of ACGP designed for on-line learning, which performs much better.

In section 2, we point out limitations of both GP and CGP. In section 3, we review the ACGP methodology and the heuristics it is capable of processing. We demonstrate ACGP's capabilities with off-line learning, and then we show its own limitations for on-line learning - using the multiplexer and the Santa Fe trail problems. We then present the updated ACGP methodology designed for on-line learning, and we illustrate with some results. We close by discussing the remaining weaknesses of ACGP.

All illustrative experiments are averages of 10 independent runs, using the standard settings for these problems, population 500, crossover 0.85, reproduction 0.05, mutation 0.05 and uniform mutation 0.05, and tournament selection size 7.

2. GP AND CGP LIMITATIONS

2.1 GP and its limitations

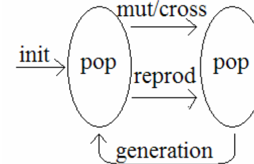

Figure 1. GP in action.

The GP's generation loop is illustrated in figure 1. The population is initialized using the available functions and terminals, mutation and crossover produce new chromosomes, while reproduction copies chromosomes into the new population. The initialization is completely random, and thus the initial trees are randomly distributed in the search space (even though not all regions of the search space are sampled the same, as according to [10]). Selective pressure, used in mutation, crossover, and reproduction, cause convergence toward higher-fitness regions. This convergence becomes the driving force behind GP – it is the primary information extracted and processed by the algorithm. However, this information is completely disregarded by mutation, which generates subtrees from the random distribution. Moreover, crossover, the primary GP operator, also uses this information to a limited extent only. While crossover's subtrees are taken from the converging population, and thus use the information expressed in the converging population, the actual method of producing offspring places these subtrees in random context, and thus again disregards the collected information again. This results in the mostly destructive nature of crossover and results in bloat.

2.2 CGP, its Capabilities and Limitations

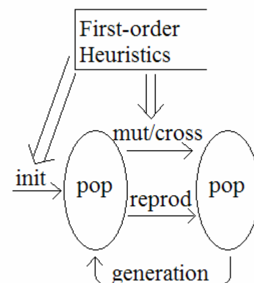

Figure 2. CGP in action.

CGP, as the other techniques for processing heuristics mentioned in section 1, is a methodology allowing the user to specify heuristics for initialization, mutation, and crossover, as illustrated in figure 2. The heuristics can be both weak and strong. The user

enters the heuristics using a specialized language [5][8]. The heuristics drive the initialization, mutation, and crossover.

CGP relies on closing the search space to the subspace satisfying the desired strong heuristics. For example, a strong heuristic prohibiting *f1()* from using *t* as its argument would never generate any tree with such labeling, neither through initialization nor mutation/crossover. Moreover, the overhead is minimal [5]. Weak constraints are processed as preferences. For example, if there is a weak heuristic saying that *f1()* is to use *t* twice as often as *s*, then *t* would show up twice as often as *s* as the argument of *f1()* in the initial population, would be twice as likely in mutation, and a subtree labeled with *t* would be twice as likely to be moved as a new subtree for *f1()* in crossover. The heuristics are all expressed as parent-child relations (and separately for the root). These relations are called *first-order* heuristics [6][7]. For example, CGP can process constraints on *f1()* and each of its arguments separately, but not as a group nor in relation to other nodes.

CGP can also use data types as the basis for its heuristics, and it supports type-overloaded functions [8]. Its processing power is illustrated in figure 3, which compares its solving capabilities against those of GP, using the multiplexer [9]. The figure shows the learning curve for GP, and for two cases of CGP when fed with two kinds of heuristics. **CGP1** uses the simple heuristic that the *if()* function should only test addresses, straight or negated. **CGP2** extends this heuristic by dropping all functions except *not()* and *if()*, and by allowing only data or recursive *if()* in the action parts of *if()*. For more examples of useful heuristics for the multiplexer, see [5][6][7].

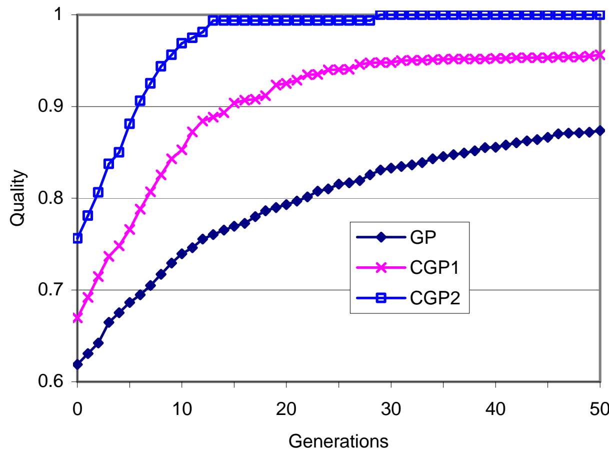

Figure 3. GP and CGP on the multiplexer: quality.

Figure 3 illustrates the usefulness of the heuristics. However, one may ask about the complexity of processing the heuristics. It turns out that due to minimal overhead [5], much smaller trees processed, and in fact savings due to reduced choices for mutation/crossover, **CGP1** and **CGP2** actually complete the 50 generations (no stopping on termination) much faster, as illustrated in figure 4 (**Total** time for the 10 runs). When we measure only the time needed for the best of 10 to find the solution, when executed concurrently (**Until solved**), the difference is much more pronounced (none of the 10 GP runs solved the problem).

However, CGP is still limited in two respects:

1. First-order heuristics are quite weak in their expressiveness.
2. The heuristics are not adjusted as GP learns more information about the search space.

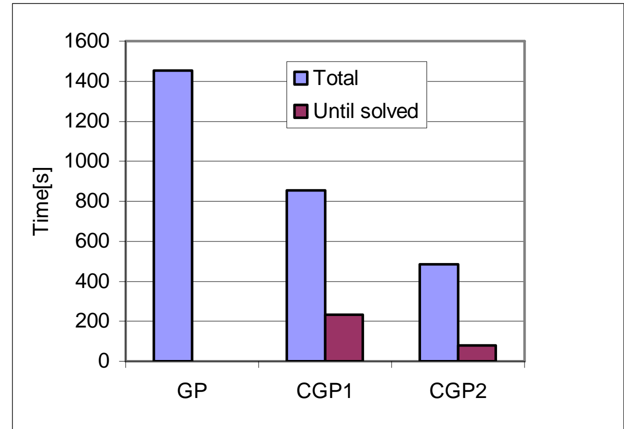

Figure 4. GP and CGP on the multiplexer: timing.

To alleviate the first problem, one must design methodologies to express and then enforce more powerful heuristics. This is quite challenging. The next version of ACGP is processing eight different kinds of heuristics, but at present it still lacks all the mechanisms needed for their enforcement.

The second problem can be explained as follow. When CGP performs mutation, it generates the new subtrees from the distribution space as defined by the first-order heuristics, but it does not adjust those heuristics as GP learns, through selection, about the relative payoff in different regions of the search space. The same happens in crossover. Thus, they both still disregard what GP learns about the search space.

3. ACGP

3.1 Off-line ACGP Capabilities and Limitations

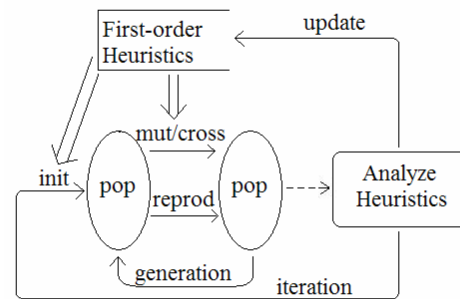

Figure 5. Off-line ACGP in action.

ACGP is a methodology to automatically update the heuristics, as illustrated in figure 5. ACGP works as CGP for a number of generations, after which it analyzes the distribution of the first-order heuristics in the population, uses this information to update

the heuristics, reinitializes the population, and starts all over. As such, it is an off-line method aiming at extracting useful heuristics about the problem. Janikow has demonstrated the capacity of the system using the multiplexer problem in [6][7]. The same capacity is again demonstrated in figures 6, 7, and 8.

Figures 6 and 7 show the quality of the best solution for the multiplexer and Santa Fe trail over 200 generations. **ACGP20** is the ACGP system in the off-line mode, while learning the heuristics every 20 generations. **ACGP20R** is the same except the population is reinitialized after the heuristics are updated. **ACGP1** is the same basic system except that the heuristics are updated at every generation. **ACGP1R** again does the same except for reinitializing the population at every generation.

ACGP20R extracts the heuristics at the end of every 20 generations, and then restarts with new population grown using the new heuristics. It clearly does best, in the long run, outperforming **ACGP20**, which after adjusting the heuristics continues from the same population. **ACGP1R** does most poorly because it does not allow selection to converge to any meaningful regions before extracting the heuristics and reinitializing the population. Thus, the heuristics tend to be random, possibly reflecting local minimum - and causing the reinitializations to be faulty. **ACGP1** does not differentiate substantially from GP itself.

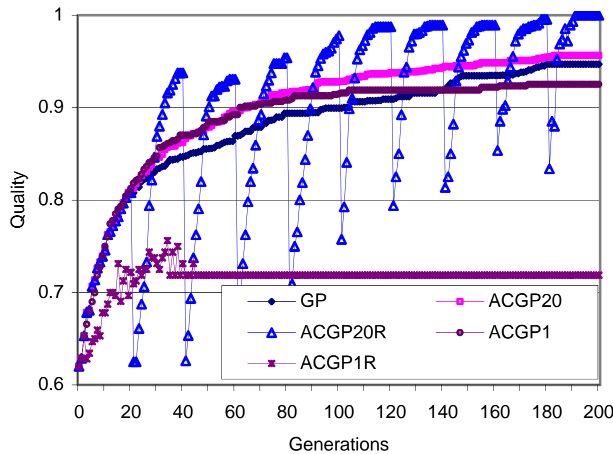

Figure 6. GP and ACGP on the multiplexer problem.

We observe similar behavior on the Santa Fe trail, as illustrated in figure 7. Here, **ACGP1R** again clearly detriments evolution, while all other systems substantially improve over GP – with **ACGP20R** coming clearly ahead.

Between figure 6 and figure 7, one may notice that the Santa Fe problem is much more difficult for the standard GP and thus we gain more from improvements from the other systems.

The results of figure 7 on the Santa Fe trail are visualized differently in figure 8. Here, we take the final heuristics, after the 200 generations, from all 5 systems and we repeat the runs for another 20 generations. As seen, the heuristics discovered by **ACGP20R** offers the most significant improvement. In fact, it starts with an almost perfect solution in just the initial population of 500, and consistently solves the problem in the next generation (averages of 10 runs saturate at the maximal value of 89).

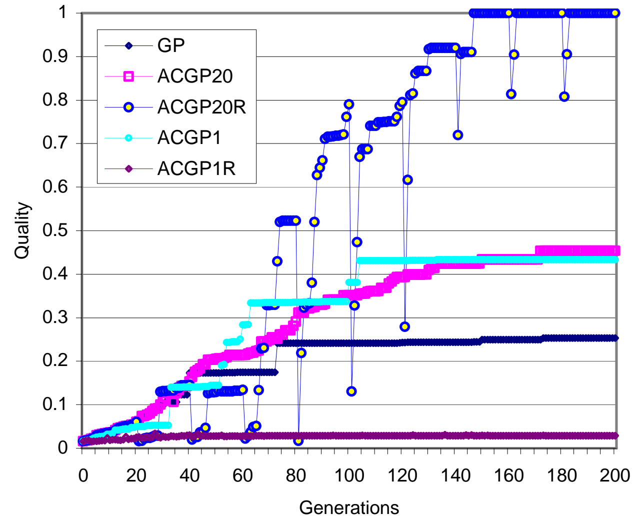

Figure 7. GP and ACGP on the Santa Fe trail.

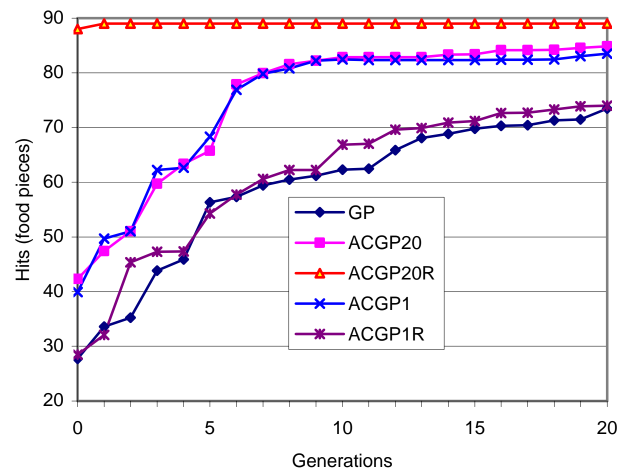

Figure 8. GP and ACGP on the Santa Fe trail: using learnt models (in terms of hits).

3.2 On-line ACGP Capabilities and Limitations

When building the on-line version, the primary challenge was to allow ACGP both the exploitation accomplished through mutation and crossover, while also allowing it to construct the first-order model. As illustrated in figures 6, 7, and 8, regrowing the population at short generation intervals has a detrimental effect, as it does not allow GP to converge, and thus the extracted heuristics are not correct. However, the same figures also demonstrate that regrowing the population provides far superior results in off-line learning. This is easy to understand – the

population converges, the heuristics reflect this convergence, so applying these heuristics to the same population has the effect of increasing the speed of convergence. However, this can lead to premature convergence. This feedback effect is avoided when, after the heuristics reflect the findings of the converged population, a new population is regrown, allowing deeper exploration of the space – not random but guided by the heuristics. Thus, our primary objective was to allow some form of regrowing, yet prevent this from randomizing the search.

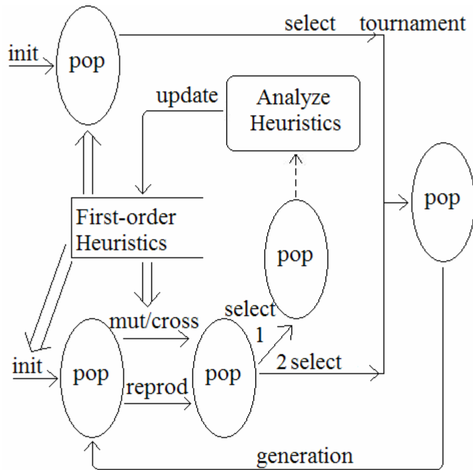

Figure 9. On-line ACGP in action.

Our first attempt was to mix mutation, crossover, with this reinitialization, allowing each to apply to a portion of the population. This produced some improvements. Our final design, illustrated in figure 9, is to use mutation, crossover, and reproduction as in CGP, then apply selection to generate samples from the higher-fitness regions, analyze these samples and correspondingly update the heuristics, use the new heuristics to generate a new population, and then merge this population with the one created by standard CGP, via tournament selection. We are still investigating the proper way to merge these two populations – tournament selections allow a controllable way of ensuring that members of both populations will be present in the new population, while at the same time using more fitted individuals to move on. Preliminary results using this on-line ACGP are presented in figures 10 and 11.

Figure 10 traces the first 20 iterations of the runs of figure 6: the multiplexer problem using GP, **ACGP1**, **ACGP1R**, **ACGP20**, and **ACGP20R**. GP and **ACGP20/20R** are identical since both of these ACGP runs do not extract the heuristics until the 20th generation. **ACGP1** is seen indifferent from GP, while **ACGP1R** is seen worse, due to its detrimental reinitializations as explained before. The figure also presents the results from the on-line version of ACGP. As seen, slowly, but surely the on-line version is capable of improving the model, and thus the run, from the very beginning. The same is seen for the Santa Fe problem in figure 11, with the same results.

The on-line model for ACGP still suffers from two problems:

1. It still relies on the relatively weak first-order heuristics.
2. The system is quite unstable with respect to many parameters.

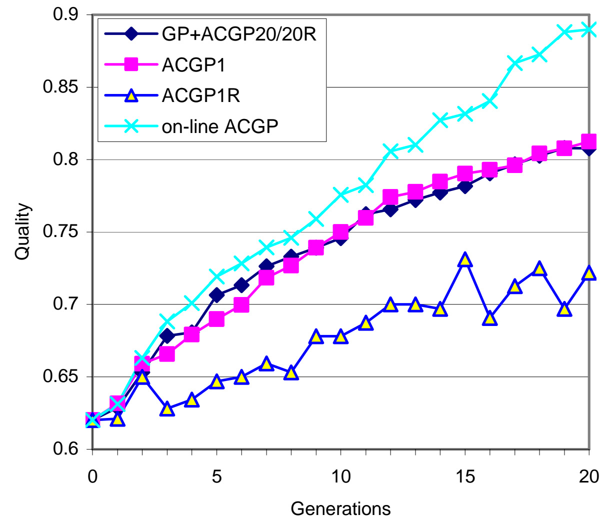

Figure 10. On-line ACGP on the multiplexer problem.

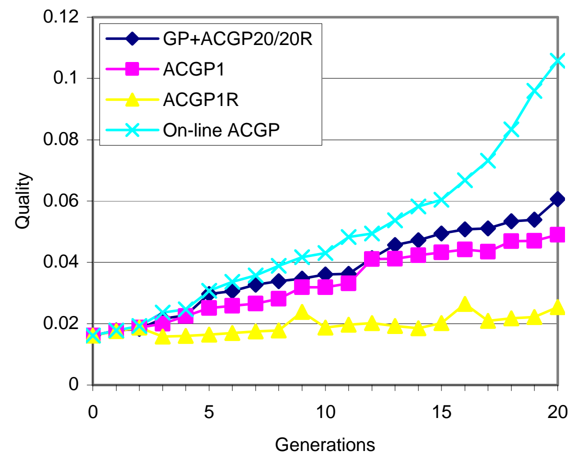

Figure 11. On-line ACGP on the Santa Fe trail.

As mentioned before, we are developing a new version of ACGP, which includes heuristics on types and overloaded functions with eight different heuristics, some of them more powerful than the simple first-order heuristics used here. As to the stability of the system, more research needs to be done.

4. CONCLUSIONS

This paper overviews the limitations of standard GP when it comes to utilizing the information collected during evolution. It then discusses how CGP alleviates some of the problems, yet faces others. Then, it presents ACGP, which in both off-line and on-line modes collects information available during GP's search into a problem model – a set of first-order heuristics. The paper

illustrates the capabilities of the off-line model, and then points out its own limitations when applied to on-line learning. Finally, it introduces the on-line ACGP model, and shows how this model improves the on-line performance.

The problems still faced by both ACGP models are: limitations of the first-order heuristics, and system instability under its parameters. We are currently working on a more powerful set of heuristics, and will continue working on the stability problem in the future.

REFERENCES

[1] Banzhaf, Wolfgang, Nordin, Peter, Keller, Robert E., and Francone, Frank D. *Genetic Programming*. Morgan Kaufmann 1998.

[2] Burke, Edmund, Gustafson, Steven, and Kendall, Graham. A survey and analysis of diversity measures in genetic programming. In Langdon, W., Cantu-Paz, E. Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M., Schultz, A., Miller, J., Burke, E. and Jonoska, N., editors. *GECCO2002: Proceedings of the Genetic and Evolutionary Computation Conference*, 716-723, New York. Morgan Kaufmann.

[3] Daida, Jason, Hills, Adam, Ward, David, and Long, Stephen. Visualizing tree structures in genetic programming. In Cantu-Paz, E., Foster, J., Deb, K., Davis, D., Roy, R., O'Reilly, U., Beyer, H., Standish, R., Kendall, G., Wilson, S., Harman, M., Wegener, J., Dasgupta, D., Potter, M., Schultz, A., Dowsland, K., Jonoska, N., and Miller, J., editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of LNCS, 1652-1664, Chicago. Springer Verlag.

[4] Hall, John M. and Soule, Terence. Does Genetic Programming Inherently Adopt Structured Design Techniques? In O'Reilly, Una-May, Yu, Tina, and Riolo, Rick L., editors. *Genetic Programming Theory and Practice (II)*. Springer, New York, NY, 2005, 159-174.

[5] Janikow, Cezary Z. A Methodology for Processing Problem Constraints in Genetic Programming. *Computers and Mathematics with Applications*, 32(8):97-113, 1996.

[6] Janikow, Cezary Z. ACGP: Adaptable Constrained Genetic Programming. In O'Reilly, Una-May, Yu, Tina, and Riolo, Rick L., editors. *Genetic Programming Theory and Practice (II)*. Springer, New York, NY, 2005, 191-206.

[7] Janikow, Cezary Z. Adapting Representation in Genetic Programming. In K. Deb et al. editors. *Proceedings of Genetic and Evolutionary Computation Conference: GECCO 2004*, 507-518.

[8] Janikow, Cezary Z. *CGP*. `http://www.cs.umsl.edu/~janikow/CGP`.

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

[10] Langon, William. Quadratic bloat in genetic programming. In Whitley, D., Goldberg, D., Cantu-Paz, E., Spector, L., Parmee, I., and Beyer, H-G., editors, *Proceedings of the Genetic and Evolutionary Conference GECCO 2000*, 451-458, Las Vegas. Morgan Kaufmann.

[11] McPhee, Nicholas F. and Hopper, Nicholas J. Analysis of genetic diversity through population history. In Banzhaf, W., Daida, J., Eiben, A. Garzon, M. Honavar, V., Jakiela, M. and Smith, R., editors *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1112-1120, Orlando, Florida, USA. Morgan Kaufmann.

[12] Montana, David J. Strongly Typed Genetic Programming. *Evolutionary Computation*, 3(2):199-230, 1995.

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

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