Second Order Heuristics in ACGP

Cezary Z Janikow
University of Missouri-St Louis
St. Louis, MO 63121
1-314-516-6352
janikow@umsl.edu

John Aleshunas
University of Missouri St Louis
St. Louis, MO 63121
jja7w2@umsl.edu

Mark W Hauschild
University of Missouri-St Louis
St. Louis, MO 63121
1-314-972-2419
mwh308@umsl.edu

ABSTRACT
Genetic Programming explores the problem search space by means of operators and selection. Mutation and crossover operators apply uniformly, while selection is the driving force for the search. Constrained GP changes the uniform exploration to pruned non-uniform, skipping some subspaces and giving preferences to others, according to some heuristics. Adaptable Constrained GP is a methodology for discovery of such useful heuristics. Both methodologies have previously demonstrated their surprising capabilities using only first-order (parent-child) heuristics. Recently, they have been extended to second-order (parent-children) heuristics. This paper describes the second-order processing, and illustrates the usefulness and efficiency of this approach using a simple problem specifically constructed to exhibit strong second-order structure.

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, Heuristics, Search Space.

1. Background
Genetic Programming (GP) is an evolutionary computation method bringing together concepts from computer science and nature. It solves a problem at hand by using a population of candidate solutions, represented as chromosomes, and by manipulating the solutions via simulated mutation and crossover – while driven by selection to explore better solutions.

Even though GP methods have been devised to work with a broad range of possible representations for the candidate solutions, the most common representation is that of a tree [1,6]. These trees are labeled with functions and terminals representing problem-specific elements: functions, connectors, constants, sensors, *etc*. The actual search space, called *genotype* space, searched by GP is uniquely determined by the labels, and only constrained by limits on tree size or depth – the trees can be labeled in any arity-consistent manner (the *closure* property [6]). The corresponding solution space, called *phenotype* space, depends on the interpretations of the labels – the interpretations provide a mapping from the search space to the solution space or from genotype to phenotype space. Somewhere in the search space, GP attempts to find a point mapped to the actual solution in the solution space, which will provide the solution to the problem at hand. The quality of a single point in the search space is determined by evaluating the mapped solution through a provided black-box fitness function.

There are some important issues to consider when designing GP, similar to those of other evolutionary methods yet specific to GP. If a given solution does not have a search space point mapped into it, it will never be discovered. Therefore, the mapping must be *onto*. To accomplish this, in the absence of detailed information about the problem or solution, the search space needs to be enlarged (a part of the *sufficiency* principle [6]). This leads to *many-to-one* mappings, with large redundancy in the representation. To handle these problems, some properties need to be there, among them many-to-one mappings to the better solutions and *proximity* induced by the mapping – if two solution points are similar in quality, they should be mapped to from neighboring points in the search space [21].

Specifically in GP, sufficiency leads to huge redundancies, while often lacking the proximity. Moreover, the large search space also generally reduces the search efficiency [2,4]. To answer these challenges, a number of methods have been proposed that ultimately prune, or reduce the effective search space, such as STGP, CFG-based GP, etc. [1].

Constrained GP (*CGP*) is another such method. It allows certain constraints on the formation of labeled trees – constraints on a parent and one of its children at a time, also called *first-order heuristics* [4] (CGP also supports restrictions based on types, along with polymorphic functions). The constraints are processed in a *closed search space* by operators with minimum overhead [2] – closed search space refers to generating only valid parents from valid children. The heuristics in CGP can be *strong*, that is conditions that must be satisfied, or *weak* expressed as probabilities. Such local probabilities effectively change the density or uniformity of the GP search space, and as such it affects the proximity of the genotype and the phenotype. CGP has been proven very successful on a number of standard GP problems when using the strong constraints only [3,4,5].

One problem facing CGP is that it requires the user to know the heuristics – CGP only provides means of adjusting the search space based on the heuristics. However, even though knowing proper heuristics can lead to great efficiency gains, the process of finding such heuristics can be very slow and inefficient [5]. *Adaptable CGP* (*ACGP*) was developed to automate the process of discovery of such useful heuristics, and the method was also shown to efficiently learn and apply the heuristics, as for example illustrated for the multiplexer problem [4].

The idea of restricting the GP search space has a long history. McPhee with Hopper [20], and Burke [16] analyzed the effect of the root node selection on GP. Hall and Soule [18] have performed even more extensive study of this phenomenon. They 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 [17], indicating that these later generations search a smaller subspace of the search space. Moreover, Langdon has shown that GP typically searches only a well defined region of the potential search space [19]. Hall and Soule call these phenomena the design evolved by GP, which process in fact resembles *top-down design* strategy [18]. However, more needs to be done about studying the effect of imposing specific designs on GP, or automatic discovery of such designs in particular. CGP and ACGP does that, working not only at the root, but working locally as well.

Estimation Distribution Algorithms (EDA) is another approach to deal with these design or more general structure issues at the probabilistic level [15], as are grammar-based methods [14] and semantic optimization methods [13]. These methods attempt to build probabilistic models, which in turn can be used to generate solutions. ACGP differs from EDA as it builds very local models (first-order), which makes it very efficient and thus effective. Moreover, ACGP uses two kinds of such models: *global* – tied to specific positions in a tree, and *local* – independent of positions in the tree, and ACGP uses the models within a standard GP search. It is more obvious to see that some heuristics near the root node help solving a problem, but it is surprising that very local heuristics can accomplish even more for seemingly complex problems [4].

Recently, the ACGP methodology has been extended to more complex heuristics – between parent and all of its children, called *second order*. These heuristics are much richer, able to express much more information. The methodology and its implementation have been extensively validated to ensure that not only they are correct but also that they operate comparably to the first-order methodology, for better comparison. However, the question whether second-order processing can be more powerful than the first-order has yet to be answered - each second-order heuristic is already processed implicitly in the first-order ACGP, by processing its first-order components. Our first attempts to answer the question using multiplexer and artificial ant indicated "no" – but careful problem analysis revealed that the actual explicit second-order structures did not differ from the implicit ones. Therefore, we have constructed an artificial problem with easily controllable strong explicit second-order structure that cannot be predicted from its first-order components. This paper introduces the ACGP second-order processing, uses the simple function to illustrate that "yes" indeed second-order processing is both more effective and more efficient.

2. ACGP and Second-Order Heuristics

2.1 Heuristics in ACGP

Heuristics in Artificial Intelligence are considered to be chunks of information, or rules-of thumb, that can lead to some improvements in knowledge or in processing. In ACGP, heuristics are probabilities attached to certain very local labeled structures. First-order heuristics are probabilities of certain parent-one-child structures, such as the probability that the binary function '+' will have '+' as its left argument – as illustrated in Figure 1a. Second-order heuristics are probabilities of certain parent-all-children structures, such as the probability that the binary function '*' will apply simultaneously to two 'y's – as illustrated in Figure 1b. One may revert this terminology to *zero-order* heuristics, that is just label probabilities – but such probabilities are superseded by first order heuristics, and one may extend to higher-order heuristics where a node is considered with its children and their children simultaneously, etc.

The heuristics are very useful in guiding the GP search. For example, if the structure as labeled in Figure 1b has high importance or usefulness, and some tree is being mutated with '*' to label a node, then the two children of that node would have higher chances of being simultaneously labeled 'y' and 'y'. The same would happen in crossover – if the first order heuristic from Figure 1a has high usefulness, the tree in Figure 1 is chosen for crossover, and the root's left subtree is chosen as crossover node, CGP would favor bringing subtrees starting with '+' from the other parent – because '+' "prefers" having '+' as its left child.

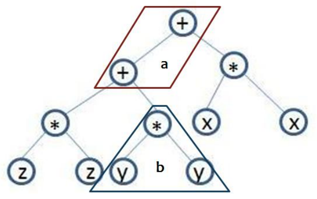

Figure 1. Illustration of a) global first-order and b) local second-order heuristics.

In ACGP, there are two kinds of heuristics. Global heuristics are position-specific as they provide information starting the root node – for example, Figure 1a is a global heuristic. Local heuristics are position-independent and they can be applied "anywhere" in the tree – Figure 1b is an illustration of local heuristics.

2.2 Discovery of Heuristics in ACGP

The building block hypothesis asserts that evolutionary processes work by combining relatively fit, short schema to form complete solutions [6]. However, small substructures cannot be easily evaluated. ACGP uses the assumption that building blocks, or structures, that occur more frequently in the fittest members contribute to the fitness of those solutions and are therefore fit building blocks. Therefore, in ACGP, the method for discovery of heuristics is straightforward – the heuristics are discovered by analyzing the best performing trees for most often occurring patterns-structures. This process does not take place after every generation as it has been shown that more time is needed for the emergence of such structures and to reduce conflicts between heuristics from different redundant representations. Instead, this happens after a number of generations, usually between 10 and 25 [4], called an *iteration.*

In addition to using multi-generation iterations, ACGP also adjusts its heuristics from the observed frequencies, rather than greedily using the frequencies as its heuristics – empirical results show that heuristics applied too greedily can lead to premature convergence into a search subspace which is incapable of representing the sought solution [4]. This is also due to the fact that GP, given its large label set, searches a space of many redundant representations and early heuristics tend to conflict between these representations. Once the search begins to converge to a specific solution and thus into specific representation, the heuristics are more reliable as a set. The *slope-off* method allows the discovered frequencies to replace a fixed part of the guiding heuristics (with 100% replacement this becomes completely greed approach). The *slope-on* method uses the frequencies to replace a growing portion of the previous heuristics, proportional to generation number.

Another method used in ACGP to increase the reliability of the emerging heuristics is to run independent smaller-population runs simultaneously and to select best heuristics from the independent set.

2.3 Representation of Heuristics in ACGP

ACGP computes substructure frequencies and represents the heuristics in tables, eventually translated into so called mutation tables. Table-representation allows for indexed random access and thus fast retrieval of information. Moreover, ACGP separately maintains its global heuristics from its local heuristics. The minimal size for the tables is completely dependent on the size of the function set *F*, the terminal set *T*, and the arity of each function, and it is shown below for the first-order heuristics:

$$\left(\left(\sum_{i}^{F_i} arity_{F_i}\right)+1\right) * (|F|+|T|)$$

The constant 1 is added to account for the global heuristics, which at the root are only maintained at the zero order. In the absence of any initial heuristics, such probability tables are initialized uniformly for every possible heuristic. The heuristics can also be initialized non-uniformly using the input interface. When ACGP analyzes the heuristics, it counts the frequencies for building blocks appearing in the fittest population members, and adjusts the probabilities of those heuristics after each iteration according to either slope-on or off schedule. The heuristics discovered in ACGP are used in crossover, mutation, and a new operator, *regrow* – a reinitialization operator used by ACGP to start each new iteration. In GP, generations build on top of each other. However, the search also converges into some subspaces. When ACGP extracts its heuristics at each iteration, it prefers to reinitialize the population according to the new heuristics in order to allow more exploration using the newly discovered heuristics, leading to better overall performance [4].

The newly discovered heuristics effectively change the space being search by GP – the search space becomes non-uniform, or the proximity between genotype and phenotype is dynamically adjusted. As shown before, this results in much more efficient search while examining smaller number of trees [4,5].

Recently, ACGP expanded the heuristic analysis and methodology to consider second-order heuristics, as shown in Figure1b. However, this also leads to more overhead both in time and space. The main reason is that the number of heuristics grows exponentially with increasing order. The equation below shows how many second-order heuristics are needed. In this case, the global heuristics are truly second order and thus we multiply the global factor rather than add. Moreover, function arity is in the exponent to account for all potential children combinations.

$$2 * \sum_{i}^{F_i} (|F|+|T|)^{arity_{F_i}}$$

For a simple problem illustrated here, with 4 binary functions, 3 variables and 11 constants, the number of heuristics computed from the above equations grows from 162 for first order, to 2592 for second order, and it would grow to about $1.37\text{x}10^7$ for third order if implemented.

Another important ACGP property is that it includes tree size in determining the best trees from the population when it comes to counting heuristics. This feature was added to dampen heuristics coming from tress unnecessarily large for their fitness – ACGP uses two-key sorting, the first key is the fitness, while the second key is tree size and it applies whenever two trees are in an equivalence class based on similar fitness. Alternatively, ACGP can also skip counting subtrees which have not contributed to fitness evaluation.

3. Empirical Illustration

3.1 Empirical Problem and its Expected Heuristics

Every second-order heuristic can be implicitly constructed from its first-order components, and thus every such second-order heuristic is already processed in the first-order system. For example, assume a problem where binary multiplication is discovered to apply to nothing but x as its left child, and nothing but y as its right child. Therefore, using just first-order processing, multiplication will always select x as its left child and y as its right child, and therefore will select x, y as its children – a second-order structure. Therefore, to compare first-order processing vs. second-order processing we need a function where at least some second-order structures cannot be implicitly processed when running first-order ACGP. WE also want to be able to modify the function easily to weaken the difference between implicit and explicit second-order heuristics. The initial function chosen is the three-variable parabolic *bowl3* = x*x + y*y + z*z.

The second-order structure apparent in this problem is that multiplication has to apply to same-variables simultaneously, a fact which cannot be implicitly constructed from observing just one child at a time. The bowl3 function has two basic minimal solutions, as seen in Figure 2. Adjusting for different permutations at the leaves, there are 6 permutations of the left tree and 6 permutations of the right tree, giving 12 minimal solution trees.

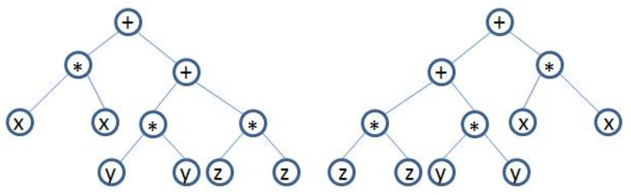

Figure 2 $x^2 + y^2 + z^2$ represented as two different trees.

By analyzing the minimal solution trees (and their permutations), we can see that even though the two trees are different in top-level structure, the local (not root) second-order heuristics are exactly the same in both:

‘*’ applies to same-variables simultaneously, and

‘+’ applies to two ‘*’ simultaneously.

While the latter heuristic can be deduced from first-order heuristics (‘+’ applies to ‘*’ on the left, and separately ‘+’ applies to ‘*’ on the right), the former heuristics cannot be possibly constructed from first-order structures and thus will require extraction and processing of second-order heuristics to possibly improve over what first-order heuristics could accomplish. Therefore, these are perfect heuristics to determine if ACGP can improve while operating on second-level structures, over first-order structures.

As mentioned, the global second-order heuristics, on the other hand, are different in the left and the right trees:

‘+’ applies to ‘*’ and ‘+’ in the left subtree, and

‘+’ applies to ‘+’ and ‘*’ in the right subtree.

Because the heuristics are opposite, there will be a conflict when combining global heuristics from multiple trees (some trees can come from the left family, others from the right family), and we can expect the initial search for heuristics to suffer until one of these families, or representations, takes over the population while using heuristics to further speed up this take-over at that time.

To add some level of complexity to this artificial problem, we enlarged the function to four binary operators {+,*,/,-}. We also enlarged the terminal set to fourteen elements by including, in addition to required {x,y,z}, eleven integer constants between -5 and 5. None of these additional functions or terminals is needed in the optimal solution.

3.2 Experimental Setup

Unless otherwise noted, all experiments were conducted as follow:

Target Equation: x*x + y*y + z*z

Function set: {+, -, *, /} (protected divide)

Terminal set: {x, y, z, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5}

Population size: 500

Generations: 500

Operators: crossover 85%, mutation 10%, selection 5%, regrow 100% at each iteration

Number of independent runs: 30

Fitness: sum of square errors on 100 random data points in the range -10 to 10

Iteration length: 20 generations

When tracing fitness, the best solution from the 30 independent runs was averaged.

3.3 Problem Solving Results

The first experiment was to compare the learning curves, that is the quality of the best solution found per generation, for a standard GP run, called Base, and for two ACGP runs while learning heuristics and applying them by updating 50% of the previous heuristics with the currently observed frequencies. The results are presented in Figure 3.

As seen, the GP-base run cannot solve the problem with better than about 10% quality. However, both ACGP runs can raise this quality to about 60-70%. On the other hand, there is little advantage in processing second-order heuristics here, while – as we already know - this problem has some second-order heuristics which cannot be constructed from their first-order components. The apparent reason for this, and for ACGP’s inability to solve the problem beyond 70%, lies in the heuristics being applied too greedily – as previously noted, the early heuristics are not reliable as some tend to be missing and others can come from conflicting representations [4]. Therefore, in the next experiment, we allow the heuristics to change slowly at the beginning.

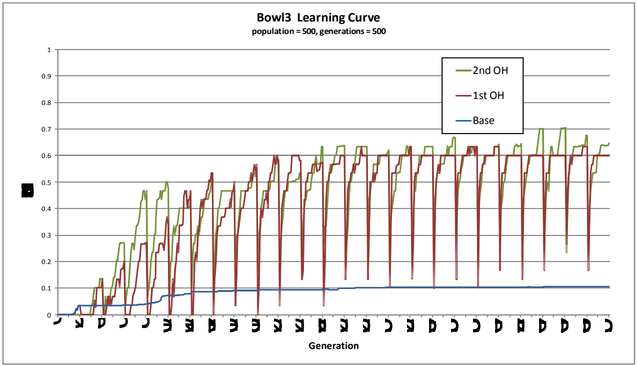

Figure 3. Comparison of GP-Base, ACGP with 1st order and 2nd order heuristics. Training slope-off with 50% update.

Another worthy observation from this experiment is that the population reinitialized with the regrow operator at each iteration tend to eventually provide better quality solution on the first generation (of a given iteration, the “dips” in the figure), due to random sampling not from the original uniform search space but from the discovered non-uniform space.

The second experiment followed the first one except that the heuristics were updated less greedily – using the slope-on method. The results are presented in Figure 4. As seen, ACGP with first-order heuristics cannot solve the problem better than with about 75% quality. On the other hand, ACGP with second-order heuristics apparently is able to discovered the strongly present second-order structure and solve the problem with 100% quality after about 10 iterations.

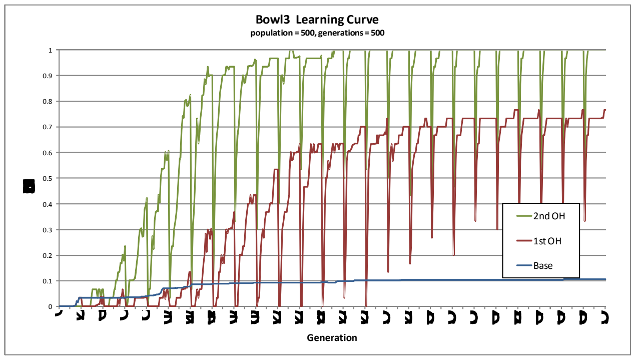

Figure 4. Comparison of GP-Base, ACGP with 1st order and 2nd order heuristics with training slope-on.

Another observation from Figure 4 is that even though both ACGP runs can clearly outperform the GP run performing uniform search, this advantage does not exists in the initial few iterations. This fact is better illustrated in Figure 5. As seen, all three runs are identical in the first iteration – before any heuristics are discovered. Moreover, it takes the first-order ACGP three iterations to outperform the standard GP. The reason is most likely the fact that initially there are the two competing representations with different global (root level) heuristics for the second-order case and different local heuristics for '+' for the first-order heuristics case, as speculated before. Once the process starts "preferring" one of the representations, possibly due to random genetic drift, the process feeds back on this and further reinforces the representation, leading to quick and dramatic improvements.

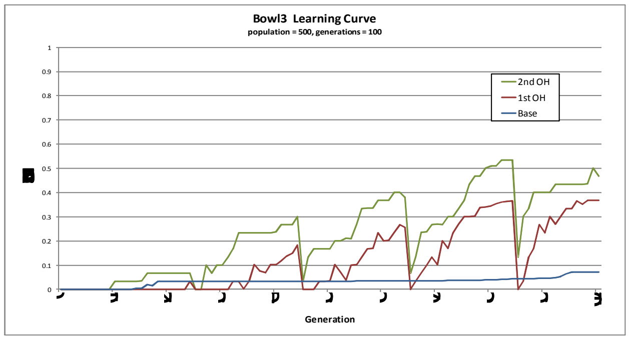

Figure 5. The first 5 iterations from Figure 4.

All previous experiments started with no heuristics (uniform operator search) and then attempted to improve the runs while discovering the heuristics. This is why the first iterations are the same for all three cases in Figure 5. However, by analyzing the problem we already know what the sought heuristics should be, both the first order and second order. Therefore, the next experiment was designed to trace the system's behavior when the heuristics are provided up front – that is when the search is completely non-uniform for ACGP from the beginning, while using the ideal heuristics from one representation only to avoid conflicts.

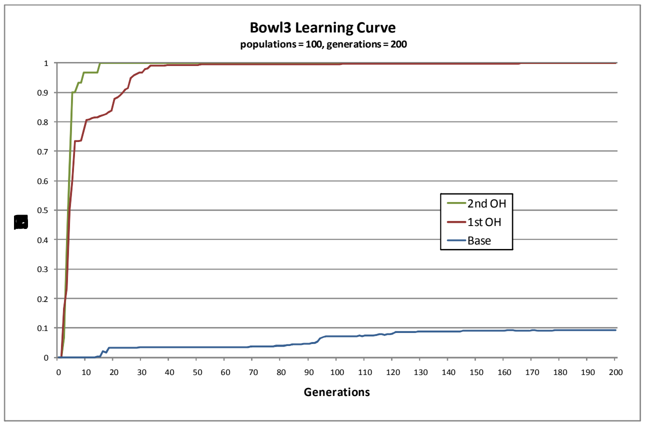

Figure 6. Comparison of GP-Base, Strong 1st Order and Strong 2nd Order Heuristics on 200 generation, no iterations.

The results are presented in Figure 6. There are no iterations here as the ACGP runs are conducted with the ideal heuristics already entered up front – ACGP allows entering first-order heuristics from the interface but the second-order heuristics were entered by modifying the code. As seen, following the early speculations, this problem has very strong set of heuristics which if already known can dramatically speed up the problem-solving process. Of course, ACGP with second-order heuristics outperforms ACGP with first order only – it requires only about 10 generations to consistently solve the problem with 100% quality while the other requires about 35.

This first experiment demonstrated the ACGP can both discover first and second-order heuristics leading to performance gains when compared generation to generation. However, if the processing overhead is too complex, this may not necessarily lead to better efficiency. To answer the question, we redraw the results from Figure 4 on time rather than generation scale.

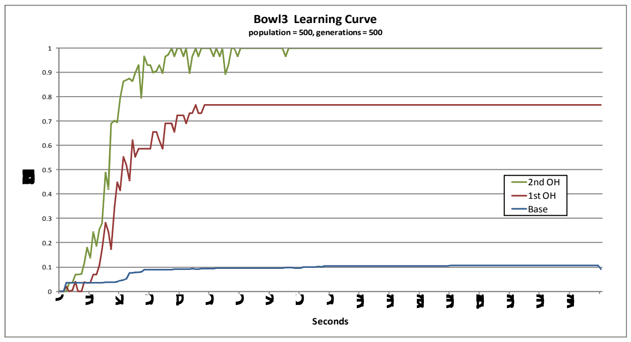

Figure 7. Comparison of Base, 1st Order and 2nd Order Heuristics on a Time Scale.

The results are illustrated in Figure 7. The apparent reinitialization dips cannot be easily seen due to averaging on time scale. The graph clearly shows that even when taking processing complexity into account, ACGP still provides a clear advantage over GP. Moreover, ACGP with the more-complex second-order heuristics still outperforms ACGP with only first-order heuristics.

These speedups of course result from finding better solutions in fewer generations, as seen in Figure 4. However, there is more into the story. When running with better heuristics, the tree sizes tend to be smaller due to the algorithm learning to avoid unnecessary and non-contributing subtrees. This can in fact amplify the efficiency gains.

Table 1. Average tree structure for Base, 1st Order and 2nd Order Heuristics.

Average	Best Tree Size	Best Tree Depth	Execution Time
Base	728.40	19.43	347.6
1st OH	123.67	10.37	44.70
2nd OH	123.87	11.40	65.67

Table 1 summarizes the best tree complexity in the three experimental cases. It indeed shows that the trees created using the first-order order and second-order heuristics contain fewer nodes and are shallower than the trees explored in the standard GP.

So far we have demonstrated that ACGP does indeed process second-order heuristics and does it very efficiently. The heuristics can be provided up front or discovered in iterations. However, the problem used so far was designed with very strong second-order structure in order to clearly validate ACGP's capability to discover and process such information. Yet, most practical problems are likely to exhibit some but not so profound second-order structure.

Therefore, the next experiment was designed to test ACGP on such cases. In this experiment, we modified the bowl3 equation as follows: *bowl3ext* = x*x + y*y + z*z + x*y + x*z + y*z. This problem still has very explicit first order structure, but the explicit second-order structure is very similar but not equal to the implicit second-order structure processed with first-order mechanisms. For example, the first-order heuristics for the multiplication are exactly the same as before, and thus the implicit seond-order heuristics are the same as before – but now they are the same as the explicit second-order heuristics. Figure 8 illustrates runs with this modified bowl3ext. As seen, ACGP has harder time discovering the heuristics now, and there is no apparent difference between processing explicit and implicit second-order heuristics.

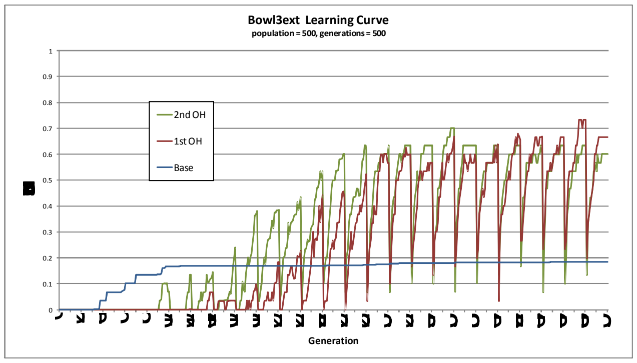

Figure 8. Comparison of Base, 1st Order and 2nd Order Heuristics with Training Slope on for an extended Bowl3 equation.

3.4 Discovered Heuristics

Another way of analyzing ACGP's performance is to look at the actual heuristics discovered after all iterations and compare them against the speculated values assumed from problem analysis.

Table 2 illustrates the first-order heuristics discovered by ACGP running in the first-order mode. The results are very close to what was speculated in Section 3.1. All heuristics start uniformly (no apriori information). The multiplication function can easily discover that it needs to apply mostly to the variables (about 72% combined out of needed 100%). The addition function also discovers that it should apply mostly to '*' and also allow association, but it clearly still cannot distinguish between the two solution families as illustrated in Figure 2 ('+' is both left and right associative while only one of them is sufficient). The reason for this confusion is that except for the '+" association, the two families have identical heuristics making it hard to distinguish between them.

The global zero-order fact that '+' should label the root node was also discovered easily. It is important to note that due to mutation and slowly updated rather than greedily computed heuristics, and some introns usually present in the trees, 100%-correct final heuristics are not expected.

Table 2. First-order heuristics discovered. Root's heuristics are zero-order.

1st Order Heuristics				
Heuristic			Initial	Final
'*'	Left arg	X	0.056	0.2289
		Y	0.056	0.2426
		Z	0.056	0.2403
	Right arg	X	0.056	0.2323
		Y	0.056	0.2489
		Z	0.056	0.2087
'+'	Left arg	'*'	0.056	0.4796
		'+'	0.056	0.2171
	Right arg	'*'	0.056	0.4168
		'+'	0.056	0.2410
Root		'+'	0.056	0.7669
Average of all other heuristics			0.056	0.0371

If we estimate the second-order heuristics from the available first-order heuristics, our estimate will be lower than needed to capture the heuristics actually present in the bowl3 equation. The first-order heuristics for the function '*' will estimate nine potential second-order heuristics {x * x, x * y, x * z, y * x, y * y, y * z, z * x, z * y, z * z}. However, we already know that bowl3 has only 3 useful second-order heuristics for '*': {x * x, y * y, z * z} and will suppress the other six heuristics. Table 3 summarizes the final second-order heuristics computed for '*' – again, the total for the three heuristics come in the 74% range out of the actual 100%.

Table 3. Second-order heuristics summary for '*'.

Multiply Heuristics				
Heuristic			Initial	Final
'*'	X	X	0.0031	0.2545
	Y	Y	0.0031	0.2368
	Z	Z	0.0031	0.2436
Average of all other heuristics			0.0031	0.0008

A similar discussion can be made regarding the second-order heuristics for '+'. The three preferred heuristics {'*' '+' '*', '*' '+' '+', '+' '+' '*'} are found after the iterations. Moreover, we can see here that using second-order heuristics ACGP is able to put preference on left-associative '+' – ACGP with first-order heuristics was not able to make this distinction as seen in Table 2.

The only dominant heuristics found for division and subtraction are a few heuristics that have no impact on the evaluation of the candidate solution. These neutral heuristics are division sub-trees that evaluate to 1 or subtraction sub-trees that evaluate to 0. An example of one of these heuristics would be (5 / 5). In other words, ACGP was able to discover that if these extraneous functions are present, they should evaluate to values easily neglected by the evaluation.

Table 4. Second-order heuristics summary for '+'.

Addition Heuristics				
Heuristic			Initial	Final
'+'	'*'	'*'	0.0031	0.3110
	'*'	'+'	0.0031	0.0688
	'+'	'*'	0.0031	0.1289
Average of all other heuristics			0.0031	0.0015

4. Conclusions

We have presented here the ACGP methodology for processing and discovery of useful second-order heuristics. This is an extension to the previously introduced and illustrated first-order ACGP, which was shown to improve search efficiency considerably on a class of standard problems. This paper demonstrates that if very strong second-order heuristics are present, ACGP is able to process them and also to discover them, does it very efficiently, and the discovered heuristics are similar to what one would expect by carefully analyzing the problem solution.

The paper also illustrated that if a problem does not have explicit second-order structure, ACGP running in first or second-order mode is the same – there are always implicitly constructed second-order structures. This is very important because it means that, in the absence of any information on second-order structures, it is not necessary to run ACGP in both modes – the second-order mode is at least as powerful regardless what heuristics are present in the domain. This feature needs to be validated with more experimentation, but it was already observed on other problems as well.

Of course, some of the local heuristics are context-specific, that is they should be different in different subtrees. ACGP relies on the simplicity of its completely local heuristics for its efficiency, but it is possible to provide some context-sensitivity – we hope to investigate this in the future.

The paper used a simple artificial problem allowing very detailed analysis of the difference between explicit and implicit second-order structures, for the sake of illustration the method's effectiveness and efficiency. The next step will be to move on to standard test or real world problems. If anything, ACGP can at least be used to predict the existence of second-order structures beyond those implicit ones. So far, we have concluded that the 11 multiplexer and the SantaFe trail problems do not have any such explicit second-order structures. If second-order processing proves beneficial on some real life problems, the next step will be to extend the method to higher-order structures.

5. REFERENCES

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

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

[3] Janikow, Cezary Z., Deshpande, Rahul, *Adaptation of Representation in GP*. AMS 2003

[4] 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.

[5] Janikow, Cezary Z., and Mann, Christopher J. *CGP Visits the Santa Fe Trail – Effects of Heuristics on GP*. GECCO'05, June 25-29, 2005.

[6] Koza, John R. *Genetic Programming*. The MIT Press. 1992.

[7] Koza, John R. *Genetic Programming II*. The MIT Press. 1994.

[8] Looks, Moshe, *Competent Program Evolution*, Sever Institute of Washington University, December 2006

[9] McKay, Robert I., Hoai, Nguyen X., Whigham, Peter A., Shan, Yin, O'Neill, Michael, *Grammar-based Genetic Programming: a survey*, Genetic Programming and Evolvable Machines, Springer Science + Business Media, September 2010

[10] Poli, Riccardo, Langdon, William, B., *Schema Theory for Genetic Programming with One-point Crossover and Point Mutation*, Evolutionary Computation, MIT Press, Fall 1998

[11] Sastry, Kumara, O'Reilly, Una-May, Goldberg, David, Hill, David, *Building-Block Supply in Genetic Programming*, IlliGAL Report No. 2003012, April 2003

[12] Shan, Yin, McKay, Robert, Essam, Daryl, Abbass, Hussein, *A Survey of Probabilistic Model Building Genetic Programming*, The Artificial Life and Adaptive Robotics Laboratory, School of Information Technology and Electrical Engineering, University of New South Wales, Australia, 2005

[13] Looks, Moshe, *Competent Program Evolution*, Sever Institute of Washington University, December 2006

[14] McKay, Robert I., Hoai, Nguyen X., Whigham, Peter A., Shan, Yin, O'Neill, Michael, *Grammar-based Genetic Programming: a survey*, Genetic Programming and Evolvable Machines, Springer Science + Business Media, September 2010

[15] Shan, Yin, McKay, Robert, Essam, Daryl, Abbass, Hussein, *A Survey of Probabilistic Model Building Genetic Programming*, The Artificial Life and Adaptive Robotics Laboratory, School of Information Technology and Electrical Engineering, University of New South Wales, Australia, 2005

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Franz Rothlauf. *Representations for Genetic and Evolutionary Algorithm*. Springer, 2010.