Impact of Commutative and Non-commutative Functions on Symbolic Regression with ACGP

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

John Aleshunas
Webster University
St. Louis, MO, USA 631119
1-314-246-7565
jalesh@webster.edu

ABSTRACT

Genetic Programming, as other evolutionary methods, uses selection to drive its search toward better solutions, but its search operators are uninformed and perform uniform search. Constrained GP methodology changes this exploration to pruned non-uniform search, skipping some subspaces and giving preferences to others, according to provided heuristics. The heuristics are position-fixed or position-independent and are just preferences on some specific labeling. Adaptable Constrained GP ACGP is a methodology for discovery of such useful heuristics. Both methodologies have previously demonstrated their surprising capabilities using only parent-child and parent-children heuristics. This paper illustrates how the ACGP methodology applies to symbolic regression; demonstrate the power of low-order local heuristics, while also exploring the differences in evolutionary search between commutative and non-commutative functions.

Keywords

Genetic Programming, Symbolic Regression, Heuristics

1. Background

Genetic Programming (GP) is a problem solving method merging ideas from nature with computation. GP solves problems by maintaining a population of viable candidate solutions, mimicking nature's chromosome representation, and by manipulating the solutions using simulated mutation and crossover – while driven by selection to explore better solutions. GP has been shown to provide robust solutions for problems such as evolving computer programs, designing logic circuits, solving many optimization and combinatorial problems where other solutions are not practical or unknown. GP has also been applied to function discovery, or symbolic regression [2, 7, 8].

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, 7]. Tree representation makes GP well suited for symbolic regression – the potential solutions are trees labeled with function and terminal, representing problem-specific elements: atomic functions, constants, variables. The actual search space, called the *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 [7]). The corresponding solution space, called the *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. In the genotype search space, GP looks for point(s) mapped to the actual solution (or approximate solution) in the solution space, given some problem at hand and using 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 [7]). This leads to multiple redundant mappings in the representation. To handle redundancy in genetic algorithms, some specific 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 phenotype, they should be mapped to from neighboring points in the genotype space [22]. It is believed that GP must also satisfy these properties. 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*) [2] is another such method. It allows certain constraints on the formation of labeled trees – constraints on a parent and its children [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 probabilities, or *heuristics*, effectively change the density or uniformity of the GP search space, and as such they affect the proximity of the genotype and phenotype space. 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 enter and thus know the strong constraints or the weak heuristics – CGP just provides the means of adjusting the search space based on the inputs. 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 discover and apply the heuristics [4, 5].

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 [19] concluded that the choice of the root node had 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 [18], 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 [20]. Hall and Soule call these phenomena the design evolved by GP, which process in fact resembles the top-down design strategy [18]. These ideas concentrated on strong constraints rather than probabilities.

Estimation Distribution Algorithms (EDA) is another approach to deal with these design or more general structure issues at the probabilistic level [16], as are grammar-based methods [15] and semantic optimization methods [16]. These methods attempt to build probabilistic models, which in turn can be used to generate solutions. ACGP differs from EDA as it builds an imperfect model using only very local information, which also makes it very efficient and thus effective. It is surprising that very local position-independent heuristics can accomplish even more for seemingly complex problems [4].

The original ACGP methodology only used *zero-order* heuristics for the root (so called *global* heuristics due to fixed location in the tree; zero-order global heuristics is just frequency of labels in the root regardless of the children) and *first-order local* heuristics (parent-one-child, position-independent). Recently, the methodology has been extended to more complex heuristics – between parent and all of its children, called *second-order*, for both global and local positions [6]. These heuristics are much richer, able to express much more information including more detailed context information for some labelings.

GP has been used for symbolic regression from the very beginning [1, 7, 8], but ACGP applications in this domain have not been reported much. This paper illustrates ACGP applicability to symbolic regression, paying specific attention to differences between first-order and second-order processing and between commutative and non-commutative functions.

2. ACGP and Low-Order Heuristics

2.1 ACGP

Constrained GP (CGP) is a methodology for processing constraints in GP [2]. The constraints are either strong or weak. Strong constraints are those required to be satisfied, while weak constraints are preferences, or heuristics, are of particular interest in this work.

Constraints are either global or local. Global constraints apply to a specific position in the tree, and due to limitations of the constraints in practice apply only to root nodes and possibly their children. Local constraints apply anywhere subject only to some local context.

In CGP, constraints are very low-order and mostly local, for reasons of efficiency rather than expressiveness [2]. There are only three classes of heuristics. Zero-order heuristics or probabilities of certain labels appearing in the tree – in practice, these heuristics are only used for root constraints. First-order heuristics are constraints on parent-one-child labelings. Second-order heuristics are constraints on parent-all-children labelings.

The heuristics are processed in a closed search space. This means that no trees are ever generated that would invalidate the constraints. There is a special initialization method that guarantees only valid trees, and mutation and crossover always guarantee to preserve the constraints if using valid parents, with minimum overhead [2].

CGP has been shown to dramatically improve GP performance even with only strong constraints and without heuristics [2,5] but the process of discovering such useful constraints can be lengthy even for the strong constraints [5] and prohibitive for weak constraints. Adaptable CGP (ACGP) was thus introduced to automate the process for discovery of useful heuristics [4]. Originally proposed for zero- and first-order heuristics, ACGP has been extended to second-order heuristics as well [6].

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, 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.

The heuristics are very useful in guiding the GP search. For example, if a tree is mutated in the left child of its root, which root happens to be labeled with '+', and the heuristic in Figure 1a has high probability, then the left child is more likely to be labeled with '+' again. This is an illustration of global first-order heuristic. If the mutation is not near the root, and the mutated node has '*' as its parent and 'y' as its left child, and the heuristics in Figure 1b has high probability, then the right node is most likely to be labeled with 'y' again. This is an illustration of local second-order heuristic.

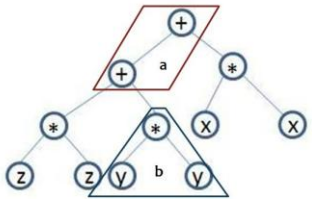

Figure 1 Illustration of a) global first-order and b) local second-order heuristics

2.2 Heuristics in ACGP

The building block hypothesis asserts that evolutionary processes work by combining relatively fit, short schema to form complete solutions [7]. The problem is that small substructures cannot be meaningfully 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 the most often occurring patterns, or 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, this is called an *iteration* [4].

In addition to using multi-generation iterations, ACGP also updates 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 disregarding some heuristics [4].

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.

ACGP uses tables to keep track of its heuristics, separately for global and local structures. Table representation allows for constant-time access and provides runtime efficiency [2]. The table size is dependent on the function set *F*, terminal set *T*, and the arity of each function. For zero- and first-order processing the size is calculated below. The constant 1 is added since ACGP uses only zero-order global heuristics even when running with first-order heuristics.

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

The heuristics are updated at runtime, but can also be pre-initialized non-uniformly using an input interface. The heuristics discovered in ACGP are used in crossover, mutation, and a new operator, *regrow* – an operator used by ACGP to start the new iteration with freshly initialized population. However, since the re-initialization uses the newly discovered heuristics, the new iteration starts with population of much higher quality than the iteration before did [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 gets modified. As shown before, this results in much more efficient search while examining smaller number of trees [4, 5].

When using second-order heuristics, the number of heuristics, and thus table sizes, grow much larger, as shown below (in this case, global heuristics are second-order)

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

For a simple problem illustrated here, with 4 binary functions, 3 variables and 11 constants, the number of heuristics computed from the above equations ranges from 162 for first-order to 2592 for second-order, and it would grow to about 1.37×10^7 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 Study on Symbolic Regression

3.1 The Problem and its Expected Heuristics

To evaluate ACGP processing capabilities with second-order heuristics with both commutative and non-commutative functions, we constructed a problem with strongly exhibited and easily controlled second-order structure and we used only commutative functions at first. The problem is the three-variable parabolic bowl.

$$Bowl3 = (x * x) + (y * y) + (z * z) \quad (3)$$

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 just the first-order heuristics. The *Bowl3* function has two basic minimal solutions (resulting from commutativity of '+'), illustrated 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. Of course, there are also non-minimal solutions.

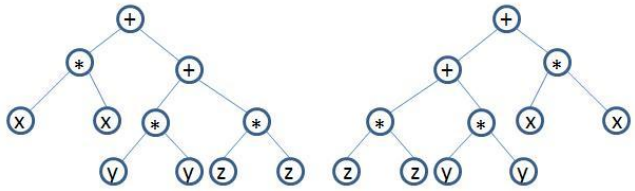

Figure 2 (x*x) + (y*y) + (z*z) represented as two different trees

By analyzing the minimal solution trees and their permutations, we can see that even though the two trees are different in the global (root) second-order structure (they are flipped), the local second-order heuristics are exactly the same in both:

'*' applies to same-variables simultaneously, and

'+' applies to two '*' simultaneously.

Assuming only best trees as shown above, with the possible permutations, it is apparent that the latter of these two second-order heuristics can be implicitly computed from the available first-order heuristics on the same trees. One local first-order heuristic is that '+' applies to "*" on the left, and another is that '+' applies to '*' on the right. Since there are no other first-order local heuristics on '+', if we compute second-order heuristics by composition of its first order heuristics, we will have the exact second heuristic above.

However, the former of the above second-order heuristics can never be computed from the observed first-order heuristics – computed second order heuristic that '*' applies to 'x' and 'x' would have the same values as that for "x' and 'y' which is not a correct heuristic. Therefore, these two above cases can help us determine if ACGP can improve while discovering second-level structures over first-order structures (when working with first-order heuristics, ACGP implicitly processes second-order heuristics but according to second-order heuristics implied from the available first-order heuristics; the same could be stated for higher order heuristics).

When it comes to global (root) second-order heuristics, the two trees (and their permutations) conflict:

'+' applies to '*' and '+' in the left subtree

'+' applies to '+' and '*' in the right subtree

Because these heuristics are the reverse of each other, 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 its heuristics to increase the take-over process.

Analysis of the first-order and the second-order heuristics used to solve this problem is instructive in understanding why ACGP can improve the solution process for this problem and how second-order ACGP has an advantage over first-order ACGP.

The *Bowl3* problem $(x * x) + (y * y) + (z * z)$ uses the following ten explicit first-order heuristics in constructing a viable solution. The first row lists the global heuristics, and the second row lists the local heuristics. Since these are first-order parent-one-child heuristics, the subscript indicates whether the heuristic is on the first (1) or the second (2) child of the parent.

$$\{+_1, +\}, \{+_2, +\}$$
$$\{+_1, *\}, \{+_2, *\}, \{*_1, x\}, \{*_2, x\}, \{*_1, y\}, \{*_2, y\}, \{*_1, z\}, \{*_2, z\}$$

Since ACGP running with only first-order heuristics is equivalent to ACGP run with second-order heuristics as implicitly computed from its first-order heuristics (or ACGP can execute with second order heuristics while only capturing first order heuristics which are subsequently used to compute the second-order order heuristics), we compute here those implicitly produced second-order heuristics. No subscripts are needed here. The first row lists the implicit global first-order heuristics.

$$\{+, +, +\}, \{+, +, *\}, \{+, *, +\}$$
$$\{+, *, *\}, \{*, x, x\}, \{*, x, y\}, \{*, x, z\}$$
$$\{*, y, x\}, \{*, y, y\}, \{*, y, z\}, \{*, z, x\}, \{*, z, y\}, \{*, z, z\}$$

However, only six of the above implicit second-order heuristics are explicitly expressed in the *Bowl3* equation.

$$\{+, +, *\}, \{+, *, +\}$$
$$\{+, *, *\}, \{*, x, x\}, \{*, y, y\}, \{*, z, z\}$$

Thus, we should expect only those heuristics to emerge if we conducted ACGP run while extracting the second-order heuristics. The remaining seven second-order heuristics are not used in solving this problem and should be discouraged from being used. However again, if we conduct ACGP run with first-order heuristics only and explicitly or implicitly process second-order heuristics without extraction, all thirteen heuristics would be processed. The differential between the second-order heuristics needed to construct a viable solution and the second-order heuristics obtained from the explicit first-order heuristics is the information differential second-order ACGP has over first-order ACGP for this specific problem.

3.2 Experimental Setup

To increase the problem complexity (and the search space), we the following binary functions $\{+, *, /, -\}$, and in addition to the required {x, y, z} we also included eleven integer constants between -5 and 5. None of these additional functions or terminals is needed in the optimal solution.

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 for the Bowl3 Problem

The first experiment was to compare the learning curves. , These curves represent the quality of the best solution found per generation, for a standard GP run, called Base, and for two ACGP runs while discovering first-order and second-order heuristics. The heuristics are extracted after each 20 generations (an iteration). In Figure 3, the heuristics discovered in the population replace 50% of the previous heuristics (or uniform heuristics on the first iteration) on every iteration.

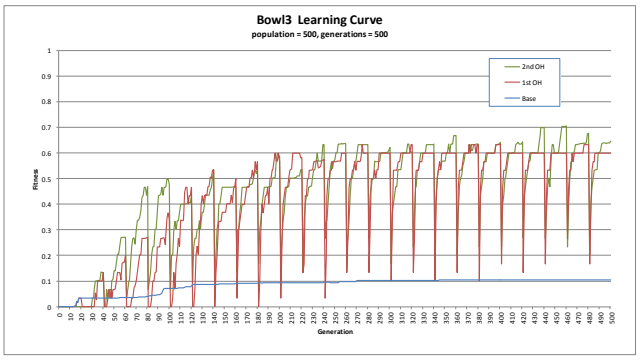

Figure 3 GP-Base, ACGP with first-order and second-order heuristics for the Bowl3 problem with greedy updates

As seen, the Base GP is not capable of solving the problem better than about 10% of the fitness. On the other hand, both first-order and second-order ACGP runs can consistently solve the problem to above 60% just after about 5 iterations (100 generations). The observed dips correspond to re-initializing the population after each iteration. It is interesting to observe that after just a few iterations the depth of the dip is above the Base case – meaning the discovered heuristics used as a model for the problem at hand provide reinitialized population of higher quality than the final population's quality in the base GP. It is also interesting to note that there is very little difference between the first-order and the second-order run, indicating that perhaps the second-order run cannot effectively utilize its potential information differential which we have seen is beneficial to solving the problem.

However, in some cases, using the heuristics too greedily (too early and too much) can lead to suboptimal convergence [4]. Since none of the GP runs was able to solve the problem, could this be due to the greedy approach taken to replace 50% of the heuristics at each iteration? Thus, in the second experiment we reduce the rate in which the observed heuristics are used to update the used heuristics – we only update portion of the heuristics proportional to generation number. That is, after the first iteration

we update only 1/25 of the initial uniform heuristics, etc. The results are presented in Figure 4. As seen, ACGP with the second-order heuristics can now solve the problem quite easily while ACGP with first-order heuristics only improves to about 70%. It seems that without the greedy update, the second-order run is now able to now utilize its information differential without being trapped with incomplete heuristics.

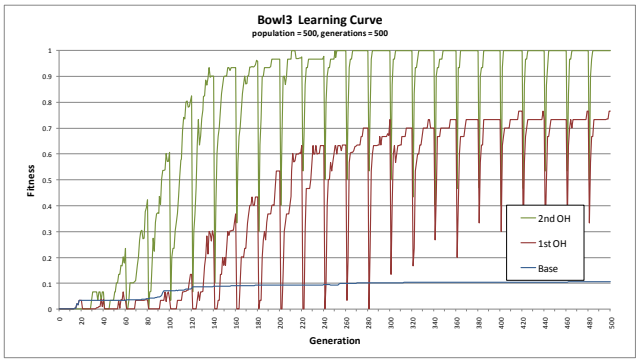

Figure 3 GP-Base, ACGP with first-order and second-order heuristics for the Bowl3 problem with less greedy updates

Another observation from Figure 3 is that even though both ACGP runs can clearly outperform the GP Base run with uniform search, this advantage does not emerge in the initial few iterations. The reason for this behavior 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, initially due to random genetic drift, the process reinforces itself and further enhances the representation, leading to quick and dramatic improvements. In other words, the ACGP run probabilistically reduces the search space and also modifies the proximity between the genotype and phenotype by modifying the heuristics driving mutation and crossover.

However, there is more to this 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 (by probabilistically reducing the search space). This can in fact amplify the efficiency gains. Table 1 summarizes the complexity of the average of the best trees for each of the 30 independent runs in the three experimental cases. It indeed shows that the trees created using the first-order and second-order heuristics contain fewer nodes and are shallower than the trees explored in the Base GP. The table also illustrates some additional time complexity for the second-order processing over first-order processing – the tree sizes are about the same yet execution time slows to almost 50% for ACGP second-order. Yes both ACGP runs handily outperform GP Base.

Table 1 Average tree structure for GP Base, first-order and second-order heuristics for the *Bowl3* problem

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

These results are interesting and illustrate a number of facts about ACGP: very low order heuristics can be useful even for symbolic regression, applying the heuristics effectively reduces the search space and thus leads to faster convergence, and the heuristics themselves are useful robust models for the problem at hand. However, one question of interest that has not been asked or answered is how the commutativity of the functions played a role in the improvements.

3.4 An Alternative Problem and its Expected Heuristics

The *Bowl3* experiment shown above demonstrates the efficacy of ACGP in discovering the necessary heuristics for a given problem. How dependent are these results on the component functions and terminals of the target solution? In particular, how the fact that the solution was made up of commutative functions only affected that solution? One could easily speculate that non-commutative function should improve ACGP reducing the conflicting heuristics that it encounters. On the other hand, one could also speculate that commutative functions are better because the local heuristics can apply the same way to both arguments of the function and provide a richer set of candidate solutions in the whole population. We decided to answer these questions and resolve the conflicting speculations using another experiment with a simple modification of the previous problem – replace the function '+' with '-'.

$$Bowl3neg = (x * x) - (y * y) - (z * z) \quad (4)$$

This modification of the problem shares most of the structural and heuristic features with *Bowl3*. Like the *Bowl3* problem, the *Bowl3neg* problem uses ten explicit first-order heuristics (here we do not separate global from local heuristics; subscripts again identify the child for the heuristic).

$$\{-_1,-\},\{-_2,-\},\{-,*\},\{-_2,*\},$$
$$\{*_1,x\},\{*_2,x\},\{*_1,y\},\{*_2,y\},\{*_1,z\},\{*_2,z\}$$

These first order heuristics combine to implicitly define thirteen second-order heuristics

$$\{-,-,-\},\{-,-,*\},\{-,*,-\},\{-,*,*\}$$
$$\{*,x,x\},\{*,x,y\},\{*,x,z\}$$
$$\{*,y,x\},\{*,y,y\},\{*,y,z\}$$
$$\{*,z,x\},\{*,z,y\},\{*,z,z\}$$

Only six of these second-order heuristics are explicitly expressed in this equation.

$$\{-,-,*\},\{-,*,-\},\{-,*,*\}$$
$$\{*,x,x\},\{*,y,y\},\{*,z,z\}$$

The remaining seven second-order heuristics are not used in solving this problem and should be discouraged. The differential between the explicit second-order heuristics needed to construct a viable solution and the implicit second-order heuristics developed from the explicit first-order heuristics constitutes information differential between ACGP running while discovering second-order heuristics and ACGP running while discovering only first-order heuristics.

Figure 5 shows the average learning curves for the *Bowl3neg* regression problem using GP-Base, ACGP with first-order heuristics, and ACGP with second-order heuristics. All three experiments were run using the same identical GP parameters as

before. The learning curves are somehow similar to those for *Bowl3*.

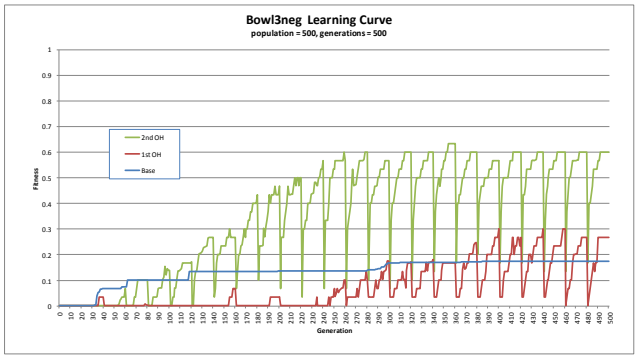

Figure 5 Comparison of GP-Base, ACGP with 1st order and 2nd order heuristics for the *Bowl3neg* problem non-greedy.

There are few apparent observations. First, neither ACGP run can solve the problem consistently. Second, the second-order run clearly outperforms the first-order run – as the differential in information has much greater impact here. Third, the first-order run has real problems on this case – this seems to support the hypothesis the low-order heuristics, without much context information, are less useful here or rather harder to extract.

Table 2 Average tree structure for GP Base, first-order and second-order heuristics for the *Bowl3neg* problem

Average	Best Tree Size	Best Tree Depth	Execution Time
Base	625.93	19.07	272.83
1st OH	140.53	11.0	46.77
2nd OH	109.0	10.63	68.9

We also compare three sizes and execution times between the runs, as before. The results are presented in Table 2. As before with *Bowl3*, trees created using the first-order order and second-order heuristics contain fewer nodes and are shallower than the trees explored in the Base GP. These results are comparable to those in Table1 so the operational performance of ACGP does not explain the difference in the search performance for the two problems. One notable difference is that the first-order ACGP maintains somehow larger trees, which is explained by its inability to find quality heuristics and thus subpar search when compared to second-order.

Next we analyze the actually discovered heuristics by the ACGP runs to further speculate on the difference in performance between the two problems.

3.5 The 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 (computed assuming minimal tree size). Table 3 illustrates the final first-order heuristics discovered by ACGP running in the first-order mode in the heuristic increment as reported in Figure 4 for the *Bowl3* problem.

The results are very close to what was speculated in the discussion above. All heuristics start uniformly (no apriori information was given) thus all initial values are the same. ACGP easily discovered that the '*' function needs to apply mostly to the variables (about 72% combined between the three variables). ACGP also discovered that the '+' function should apply mostly to '*' and also allow association (the final probabilities are about twice higher for '*' children), but it clearly still cannot distinguish between the two solution families ('+' 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.

ACGP also easily discovered the global zero-order heuristic stating that '+' should label the root node.

Table 3 First-order heuristics discovered in the *Bowl3* commutative experiment. Root's heuristics are zero-order. Only highly evolved heuristics are shown

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 summarizes the final second-order heuristics discovered in the second-order run, against those computed from available first-order heuristics, producing the information differential. The table clearly shows the advantage of extracting second-order heuristics (large differential).

Table 4 Second-order heuristics summary for '*' as discovered in the ACGP second-order run versus computed from first order heuristics

Multiply Heuristics						
Heuristic			Initial	Final	Computed	Difference
'*'	X	X	0.0031	0.2545	0.0532	+ 0.2013
	Y	Y	0.0031	0.2368	0.0604	+ 0.1764
	Z	Z	0.0031	0.2436	0.0502	+ 0.1934
Average of all other heuristics			0.0031	0.0008	-	-

A similar discussion can be made regarding the second-order heuristics for '+' and is illustrated in Table 5. The three preferred heuristics (in infix notation) {'$**+$','$*+*+$','$++*$'} are found after the *Bowl3* run.

The only dominant heuristics discovered 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.

Table 5 Second-order heuristics summary for '+' as discovered in the ACGP second-order run versus computed from first order heuristics

Addition Heuristics						
Heuristic			Initial	Final	Computed	Difference
'+'	'*'	'*'	0.0031	0.3110	0.1999	+ 0.1111
	'*'	'+'	0.0031	0.0688	0.1156	- 0.0468
	'+'	'*'	0.0031	0.1289	0.0905	+ 0.0384
Average of all other heuristics			0.0031	0.0015	-	-

The results in this table resemble those in Table 4. When the final discovered second order heuristics are compared to those computed from the final first order heuristics the information gain of the explicit second order heuristics become clear. The single exception to this observation is found in the {'$++*$'} heuristic. This heuristic and {'$**+$'} are complementary reflections and only one is needed in a final solution so this heuristic result is far from surprising. The overall rate of differential is much smaller though.

Next, we recompute these tables for the *Bowl3Neg* problem, with non-commutative function '-' instead of '+'. Table summarizes the results for '*' – they seem similar to those for *Bowl3* which is understandable since '*' is still commutative.

Table 6 Second-order heuristics summary for '*' as discovered in the ACGP second-order run versus computed from first order heuristics

Multiply Heuristics						
Heuristic			Initial	Final	Computed	Difference
'*'	X	X	0.0031	0.2333	0.0905	+ 0.1428
	Y	Y	0.0031	0.2333	0.0905	+ 0.1428
	Z	Z	0.0031	0.2333	0.0905	+ 0.1428
Average of all other heuristics			0.0031	0.0008	-	-

Table 7 Second-order heuristics summary for '-' as discovered in the ACGP second-order run versus computed from first order heuristics

Subtraction Heuristics						
Heuristic			Initial	Final	Computed	Difference
'-'	'*'	'*'	0.0031	0.3846	0.0005	+ 0.3841
	'*'	'-'	0.0031	0.0001	0.0002	- 0.0001
	'-'	'*'	0.0031	0.3846	0.2345	+ 0.1501
Average of all other heuristics			0.0031	0.0015	-	-

However, when the heuristics are summarized for the non-commutative '-' (Table 7), we can see that the differential, or advantage of second-order over first-order ACGP is much greater here (comparing the differentials in Table 7 and Table 5). This clearly explains why the *Bowl3Neg* runs with the second-order ACGP outperformed those with first-order ACGP.

Another way to explain the better performance of second-order heuristics with non-commutative functions is as follows. The principal difference between the two target equations is the component functions and the fact that one equation uses strictly commutative functions whereas the other equation substitutes a non-commutative function. Since *Bowl3* uses only commutative functions, many different structural variations will be able to form highly fit candidate solutions. Alternatively *Bowl3neg* which includes the non-commutative '-' function imposes a stricter structure of its components in the formation of a highly fit candidate solution. This difference of component functions in the target equation conditions the search for either problem. One might infer that GP regression problems with only commutative functions in their target solution find it easier to evolve solutions because a highly fit solution can be constructed in many more configurations than a target equation that includes a non-commutative function. This is an interesting conjecture and requires further experimentation to support it.

4. Conclusions

We have illustrated how the Adaptable Constrained GP (ACGP) discovers and uses valuable low-order heuristics for problem-solving, symbolic regression in this case. We have shown through analysis of specifically constructed problems that ACGP running with richer second-order heuristics has a potential information differential over ACGP running with less informative first-order heuristics. Empirical analysis illustrated that the differential is beneficial if the approach is not too greedy (which causes loss of heuristics) and that both approaches not only improve the solution found but also do that more effectively (in time and space).

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. 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.

We have also demonstrated that ACGP can benefit from its heuristics differently for commutative and non-commutative functions. In particular, second-order heuristics can capture and utilize larger information differential over first-order heuristics, resulting in faster learning.

The results are intended for preliminary illustration – more experiments and analysis is needed to have a better picture of how commutative properties affect different levels of heuristics available in ACGP.

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] Janikow, Cezary Z., Aleshunas, John, Hauschild, Mark W. Second-Order Heuristics in ACGP. In *ACM Genetic and Evolutionary Computation Conference (GECCO)* (Dublin, Ireland 2011), ACM.

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

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

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

[10] 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

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

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

[13] 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

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

[15] 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

[16] 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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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