Cost-benefit Analysis of Using Heuristics in ACGP

John Aleshunas
University of Missouri St. Louis
St. Louis, United States
jja7w2@umsl.edu

Cezary Janikow
University of Missouri St. Louis
St. Louis, United States
janikow@umsl.edu

***Abstract*—Constrained Genetic Programming (CGP) is a method of searching the Genetic Programming search space non-uniformly, giving preferences to certain subspaces according to some heuristics. Adaptable CGP (ACGP) is a method for discovery of the heuristics. CGP and ACGP have previously demonstrated their capabilities using first-order heuristics: parent-child probabilities. Recently, the same advantage has been shown for second-order heuristics: parent-children probabilities. A natural question to ask is whether we can benefit from extending ACGP with deeper-order heuristics. This paper attempts to answer this question by performing cost-benefit analysis while simulating the higher-order heuristics environment. We show that this method cannot be extended beyond the current second or possibly third-order heuristics without a new method to deal with the sheer number of such deeper-order heuristics.**

Keywords-Genetic Programming, Adaptable Constrained Genetic Programming, Building Block Hypothesis, Heuristic.

I. BACKGROUND

Genetic Programming (GP) is an evolutionary computation method merging concepts from computer science and biology. It has been shown to provide robust solutions for problems such as evolving computer programs, designing logic circuits, discovering mathematical equations, and solving other optimization and combinatorial problems where other solutions are not practical or unknown [2].

Genetic Programming represents a population of candidate solutions as trees, lists or graphs, but the dominant representation, and the one this paper concentrates on, is the tree representation [2]. These trees are composed of elements from a predetermined set of functions and terminals, called here labels. A given function can label any node whose number of subtrees equals the function arity. Terminal labels, such as constants or variables from the problem environment, label the leaves.

The search space is the space of all (up to some size limit) trees that can be labeled with the provided labels. Somewhere in the space, we should find the actual solution to the problem at hand or otherwise GP will not be capable of ever finding the solution. The quality of a single solution-tree, and therefore its "proximity" to the sought solution, is the tree's evaluation through a provided black-box fitness function.

After the initial sampling in the GP population, new solutions are evolved using genetic operators such as crossover and mutation, while guided by selection based on fitness evaluations. In crossover, randomly chosen sub-trees are exchanged between two parents. In mutation, a randomly chosen sub-tree is regrown.

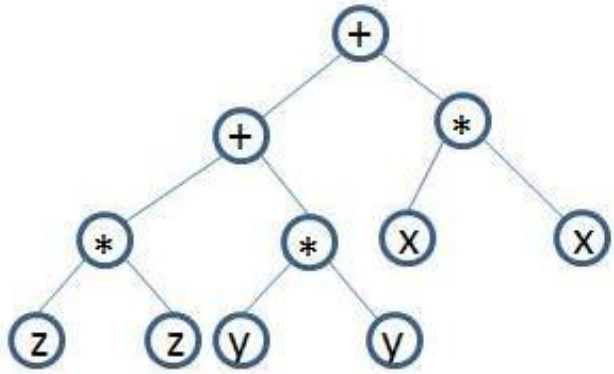

Figure 1GP solution represented as a tree for the algebraic formula x*x+y*y+z*z

One critical issue with GP design is the choice of the labels. The labels, along with limitations on explored tree sizes, determine the search space for GP (while population, selection and operators determine the means of searching the space). If the label set is not sufficient, the desired solution does not exist in the search space and therefore GP will never find it. This is often referred to as the sufficiency principle [2]. To avoid this problem, the label set is often greatly enlarged. Unfortunately, this increases the search space and generally reduces the GP effectiveness [2][4]. To answer this challenge, a number of methods have been proposed that ultimately prune, or reduce the effective search space, such as STGP, CFG-based GP, etc. [2].

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 [4] (CGP also supports restrictions based on types, along with polymorphic functions, but this paper does not use these extended capabilities). The constraints are processed in a *closed search space* by operators with minimum overhead: closed search space refers to generating only valid parents from valid children [4]. The constraints in CGP, called *heuristics*, can be *strong*, that is conditions that must be satisfied, or be expressed *weakly* as probabilities. Such local probabilities effectively change the density or uniformity of the GP search space. CGP has been proven very successful on a number of

standard GP problems especially when using the strong constraints [4,6].

CGP requires the user to know the heuristics – CGP only provides means of adjusting the search space given the heuristics. *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 [5,7]. Recently, the method has been extended to more complex heuristics. The stronger heuristics, and the method, have been validated as providing efficient speed up to evolution in cases where second-order structures exist in the problem at hand [1].

ACGP contrasts from other GP improvement techniques such as Estimation of Distribution Algorithms (EDA) for GP [10], applying grammar-based methodologies to GP [9] and semantic optimization methods [8]. Those methods attempt to build probabilities on labeling specific tree nodes rather than tree-position-independent probabilities as ACGP allows. In addition, ACGP works within a standard GP (extends *lilgp*).

In ACGP, with the growing complexity of the heuristics, come costs associated with the processing and storage of these heuristics. Second-order heuristics do not pose significant overhead; however, higher order heuristics do. The paper analyzes the cost-benefit relationship for the heuristics. Because no implementation exists beyond the second order, the benefits of higher-order heuristics are speculated, and the cost is simulated.

In this paper, we first introduce the basic principles of ACGP, its heuristics and their discovery method, followed by cost analysis. Then, we perform complete cost-benefit analysis for first and second order heuristics, tracing benefits and combing them with cost. Finally, we discuss the benefits of higher order heuristics, and measure their cost by using the current implementation to simulate cases with higher demands. We close by suggesting some ways to alleviate the cost problem.

II. HEURISTICS AND ACGP

A. 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 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 2(a). *Second order* heuristics are probabilities of certain parent-all-children structures, such as the probability that the binary function '*' will apply simultaneously to 'y', as illustrated in Figure 2(b). One may revert this terminology to *zero order* heuristics, that is just label probabilities (these are less useful except for the root node), and extend to higher order heuristics where a node is considered with its children and their children, etc.

The heuristics are very useful in guiding the GP search. For example, if the structure as labeled in Figure 2(b) has high probability, that is it is a useful structure (useful building block), and some tree is being mutated with '*' to label a node, then the two children of this node would have higher chance of being labeled 'y' and 'y' as according to this heuristic. The same would happen in crossover – if the first order heuristics from Figure 2(a) has high probability, the tree in Figure 2 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.

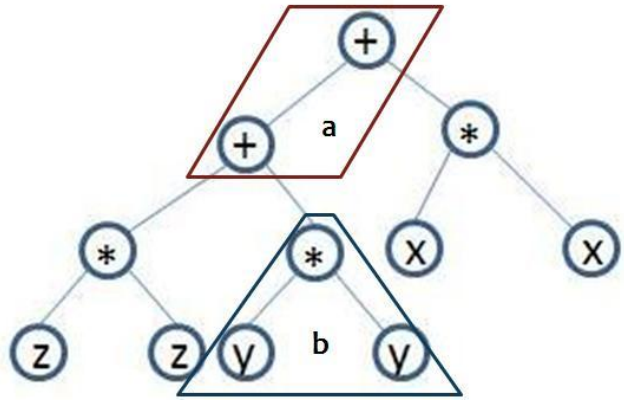

Figure 2a) First order and b) second order heuristics on trees

In EDA methods, some probabilities are maintained for specific positions in the tree [10]. In ACGP, there are two kinds of heuristics instead. *Global* heuristics are position-specific as they provide information starting from the root node. *Local* heuristics and position-independent and they can be applied in any position in the tree.

B. Discovery of Heuristics in ACGP

In ACGP, the method for discovery of heuristics is straightforward – the heuristics are discovered by analyzing the best performing trees. 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 [5]. Instead, this happens after a number of generations, called an *iteration*.

The building block hypothesis asserts that evolutionary processes work by combining relatively fit, short schema to form complete solutions [3]. The problem with this assertion is that most problems are not decomposable and it is often difficult to determine the fitness of a particular building block let alone determine its contribution to the individual's fitness. ACGP uses the assumption that building blocks, or structures, that occur more frequently in the fittest population members contribute to the fitness of those solutions and are therefore fit building blocks.

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 [5]. This is 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.

C. Representation and Cost of the Heuristics in ACGP

ACGP computes its frequencies and represents its heuristics in tables, eventually translated into so called mutation tables. Table representation allows for random access indexing 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 in Eq. 1 for the first order heuristics. 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 to uniform probabilities for every building block. The heuristics can also be initialized to non-uniform probabilities using CGP's input interface. When ACGP analyzes the heuristics, it observes the frequencies which building blocks appear in the fittest population members and adjusts the probabilities of those building blocks on each iteration.

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

The discovered heuristics in ACGP are used in crossover, mutation, and a new operator, *regrow*. Each operator needs to access the tables in order to apply as according to the heuristics. Regrow is an initialization operator used by ACGP after each iteration. In GP, generations build on top of each other. However, the search also converges into better subspaces. When ACGP extracts its heuristics at each iteration, it prefers to reinitialize the population according to the new heuristics [5]. This reinitialization tends to remove bloat, produce more compact trees from the same subspace in which the search converges, yet reduce premature convergence.

The newly discovered heuristics effectively change the space being search by GP – the space density changes, or rather the search space becomes non-uniform. In a way, the heuristics become shortcuts in the space, allowing adjustments in how genotype and phenotype relate to become more properly aligned to solve the problem in the most efficient way. As proven before, this results in much more efficient search while examining smaller number of trees [5,7]. This is often a significant improvement for a modest overhead storage and processing cost.

Recently, ACGP expanded the heuristic analysis and methods to consider second order heuristics, as shown in Figure 2(b). This has also been shown to lead to more efficient search [1]. However, this also leads to more overhead both in time and space. The main reason is that the number of heuristics grows substantially over the first order. Equation 2 shows how many second order heuristics are needed. In this case, the global heuristics are truly second order and thus we multiply the other factor rather than add. Moreover, function arity is in the exponent to account for all children combinations.

$$2 * \sum_i^{F_i} (|F| + |T|)^{arity_{F_i}} \qquad (2)$$

III. Cost-Benefit Analysis

Any improvement to any method should always be considered in the context of the cost of the method, or otherwise the overall performance may suffer. The benefit of using heuristics in GP is to speed up evolution, by utilizing the discovered heuristics which again more close align genotype and phenotype, or adjust the search to only the optimal subspace. As a secondary benefit, the discovered heuristics themselves may be invaluable information in general or for a new search, even if the cost is of obtaining them is initially high.

The cost of heuristics is the space and time requirements needed to carry out the search in the modified-density space. With efficient implementation of the closed search [4], this cost is made up of:

1. Allocating all needed counters and tables – this is done in the setup stage.
2. Time to process the better trees, analyzing and counting various strictures, and to adjust the heuristics, at each iteration – this happens infrequently, at each iteration only.
3. Time for the operators of mutation, crossover, and regrow to access the heuristics – this is performed frequently (mutation and crossover are frequent; regrow is only at iteration time).

First, we perform a detailed cost-benefit analysis up for first and second order heuristics because a complete implementation exists. In this case, we show the benefits and then combine with cost. Because no ACGP implementation beyond the second order exists, when moving to higher order analysis we extrapolate on the benefits and measure the cost by simulations.

A. Experimental Methodology

To perform this analysis, we use the function shown in Eq. 3. This controlled problem exhibits strong second order structure but less obvious first order structure – this will allow us to access the benefits of processing proper heuristics yet observe the difference in improvements attributed to different levels of heuristics. The function and terminal sets are sufficient yet not minimal as we do not want to assume that we know that much about the problem at hand. Accordingly, the function set is set to F={*, /, $^+$, -} and the terminal set is set to T={x, y, z, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5}.

All experiments are repeated independently ten times, and averages are reported. Unless otherwise noted, population size was set to 500, generations to 500, and iteration to 20 generations.

$$(x * x) + (y * y) + (z * z) \qquad (3)$$

B. Benefit of First and Second OrderHeuristics

Among the possible first order building blocks, ten local heuristics are highly desirable, as seen by analyzing Eq. 3. This set of these desired heuristics is shown below – subscript indicates child number:

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

ACGP runs as GP for 20 generations (1 iteration). At that time, it considers the best trees in the population, analyzing them and counting all found first order structures (we use literal counting; however, ACGP also allows other counting methods which disregard unexpressed subtrees and counts more frequently used subtrees with higher weights). Then, ACGP adjust the initially uniform heuristics according to the observed frequencies, reinitializes the population using the modified heuristics (using regrow), and continues its generations with crossover and mutation through the next iteration. However, mutation and crossover are now non-uniform as according to the heuristics. After several of these analysis intervals, the building blocks that contribute most to the best solutions begin to emerge in the heuristics set, and the evolution picks up.

The benefits of using such first-order heuristics over a traditional GP run can be seen in Figure 3, which shows the fitness of the best individual averaged over 30 independent runs. However, the first order heuristics, even the most desirable as shown above, clearly do not capture the true structure present in the problem. For example, the first order heuristics state that '*' can apply to 'x' as its first argument and 'x' as its second argument but are incapable of expressing the fact that this should be simultaneous as seen in Eq. 3. In particular, using the above first order heuristics, the following thirteen second order heuristics are all equally likely:

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

However, only the diagonal heuristics are truly relevant and contribute new information to solving Eq. 3. The others will not contribute to a fit solution. ACGP running with only first order heuristics can process the above heuristics implicitly, but when derived from first order heuristics all nine of the second order heuristics will be equally likely.

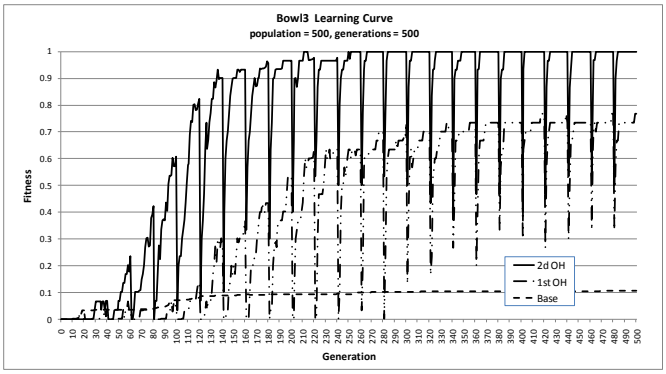

Figure 3 Comparison of Base, first order and second order heuristics in ACGP

When ACGP is executed with second order heuristics, it is capable of differentiating among the above nine heuristics, and thus it should be capable of using this to an advantage. Figure 3 shows that it indeed does so, clearly outperforming not only standard GP but also ACGP running with first order heuristics.

C. Cost of First and Second Order Heuristics in ACGP

Since all of the functions in Eq. 3 are binary functions, the total number of possible first order heuristics is computed using an adjusted form of Eq. 1, shown in Eq. 4. Given the problems settings as above, this yields 162 possible first order heuristics.

$$(2 * |F| + 1) * (|F| + |T|) \quad (4)$$

Among the 162 possible building blocks, only six local first order heuristics are highly desirable, as seen before, plus the global heuristic placing '+' in the root.

Using second order structures, the total number of heuristics can be computed using an adjusted form of Eq. 2 as presented in Eq. 5.

$$2 * |F| * (|F| + |T|)^2 \quad (5)$$

Given the problem settings, this yields 2592 second order heuristics. Among those, only three local heuristics, as indicated above, are useful, plus some global placing {+ + *} and {+ * +} in the root.

The sizes hardly seem prohibitive, but to assess all cost we must take all cost components into account – the cost involves not only the space but also accessing and updating the heuristics. Rather than directly computing the cost and then comparing it against the benefits of Figure 3, we re-plot Figure 3 on time rather than generation scale – this way all possible cost is taken into account. The re-plotted results are shown in Figure 4, extended to 180 seconds which was the longest run among the three cases. The observed dips are due to regrowing the population at interval points – however the dips are minimal due to averaging over independent runs.

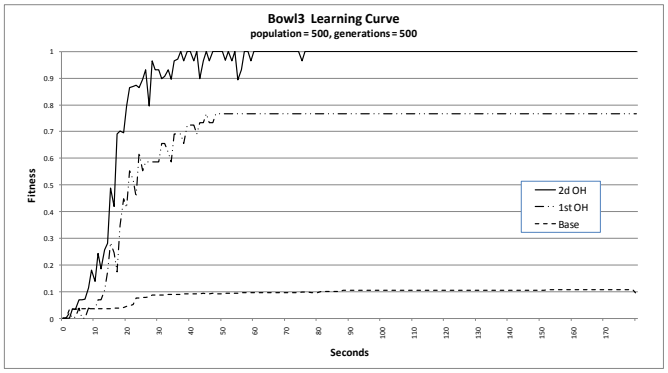

Figure 4 Comparison of Base, 1st Order and 2d Order Heuristics for Equation 3 180 seconds, 500 population

As can be seen, ACGP is able to produce the best solution while learning second order heuristics – and even the first order heuristics are a substantial improvement over the standard GP. In fact, both kinds of heuristics provide significant advantage to GP even if cost is taken into account. Second order processing outperforms first order – on the benefit side this was expected since Equation 4 exhibits strong second order structure, but showing that the benefit is sustained when taking cost into account is welcome information. Additional analysis presented in [1] shows that the advantage persists even when the second order structure is not so dominating in the problem being solved.

D. Benefits of Higher Order Heuristics

ACGP implementation for higher order structures does not exists, and therefore empirical support for the benefits of extracting and using higher order heuristics cannot be performed. However, from the benefit analysis for first and second order heuristics it is easy to speculate that if the problem exhibits higher order structures, processing such higher order heuristics should be beneficial.

E. Cost of Higher Order Heuristics

The engine that drives the performance gains of ACGP is the heuristics extracted from the problem and processed in the operators. However, the number of higher order heuristics grows very substantially, and it poses storage and processing challenges to access and adjust the heuristics. This section analyses this issue.

The cost of higher order heuristics is that of processing, and modifying the tables needed to maintain the heuristics. The first question is how big are those tables? Equation 6 shows the formula to compute the size of the tables for third order heuristics, and Table 1 shows the actual sizes for the problem illustrated in this paper for order 1-6. As seen from that table, the sizes become prohibitive around third or fourth order. Let us illustrate it in detail.

$$2 * \sum_i^{F_i} \left(\left(\sum_i^{F_i} (|F| + |T|)^{arity_{F_i}} \right) + |T| \right)^{arity_{F_i}} \qquad (6)$$

Table 1 GROWTH OF THE NUMBER OF HEURISTICS FOR DIFFERENT LEVELS

1st OH	1.62×10^2
2nd OH	2.59×10^3
3rd OH	1.37×10^7
4th OH	3.76×10^{14}
5th OH	2.48×10^{29}
6th OH	1.62×10^{59}

The tables in ACGP are generated during the setup, and subsequently are randomly indexed to access the information as needed in mutation, crossover, regrow, and are linearly accessed to adjust the heuristics at each iteration time. Random indexing is cheap as long as the language supports indexes of needed size and the total size does not exceed possible process space. The process space can be quite substantially on some 64-bit machines yet the real challenge comes from the limited RAM available on a computer system even if the system is capable of addressing a larger address space.

The empirical analysis was performed using a Dell Latitude 6400 with Intel Duo core P 8700 2.53 GHz processor with 4 GB RAM [3.48 GB effective] under Windows XP Pro SP3. This system was chosen for convenience, but it does not allow large processes and therefore we turned the simulation to gain the needed data.

Because ACGP does not provide implementation for heuristics above second order, we instead took the second order case and enlarged it at controlled intervals, generating artificial cases with controllably increasing number of heuristics – to increase table sizes. These cases are listed in Table 2. The case 1x is the previous second order case. The case 10x refers to the same problem with an artificially increased function and terminal set that was produced by duplicating the original set of labels in order to increase the number of heuristics by one order of magnitude, etc. However, because the 100,000x could not complete on our system, we used the 10,000x plus case giving a three-fold increase over the previous case. With these artificial cases, we will be able to simulate table sizes for up to third order heuristics but not up to fourth.

Table 2 EXPERIMENT COMPLEXITY LEVELS

Level	Functions (binary)	Terminals	Combinations
1x	4	14	2,592
10x	10	26	25,920
100x	28	40	258,944
1000x	62	82	2,571,264
10,000x	120	208	25,820,160
10,000x plus	188	267	77,841,400

First, we measure separately the time requirements for various operations of ACGP:

- Setup time, which includes table allocations and transformations of the internal structures
- Fitness time, which is time to evaluate the trees which should be independent of the heuristics involved
- Operator time: mutation and crossover, which will require access to heuristics
- Regrow time: reinitializing the population at each interval which requires access to heuristics
- Weight time: analyzing best trees, computing frequency tables, and recomputing the heuristics, all at iteration time
- Processor time: total time
- Other run attributes such as average tree size and depth, RAM and Virtual Memory (VM) used.

Table 3 RUN STATISTICS FOR Table 2

Seconds	1x	10x	100x	1,000x	10,000x	10,000x plus
Setup time	0.27	1.00	3.03	8.67	26.23	28.00
Fitness time	38.33	17.60	17.17	18.10	20.53	27.00
Operator time	5.73	3.00	2.93	3.50	5.13	8.00
Regrow time	0.00	0.00	0.00	1.00	7.93	16.00
Weight time	0.28	0.20	0.72	3.46	19.20	49.92
Processor time	00:37.0	00:34.1	00:31.1	01:02.5	03:01.5	07:04.2
Size	123.87	44.87	39.73	54.47	57.13	51.00
Depth	11.40	6.93	5.47	6.57	7.40	9.00
RAM (in K)	7,400	8,136	12,612	71,500	639,160	1,866,716
VM (in K)	14,756	14,756	22,948	137,644	673,844	1,871,772

The data is compiled in Table 3. The Setup time grows with increasing number of heuristics, as expected, as larger tables need to be allocated and initialized. Fitness time is not affected by increasing heuristics and is rather related to average tree sizes. However, this would not be true with increasing

mismatch between RAM and total memory required as seen later.

Operator time is stable across the cases – random access to heuristics is not affected by table sizes. Regrow time increases, which is in contrast to Operator time. Because both Regrow and Operator access the same heuristics, this must be caused by memory allocation and deallocation during regrowing the trees – this could likely be alleviated by carefully reusing the same memory. Weight time grows most substantially with growing number of heuristics – this involves traversing some trees and then updating all heuristics.

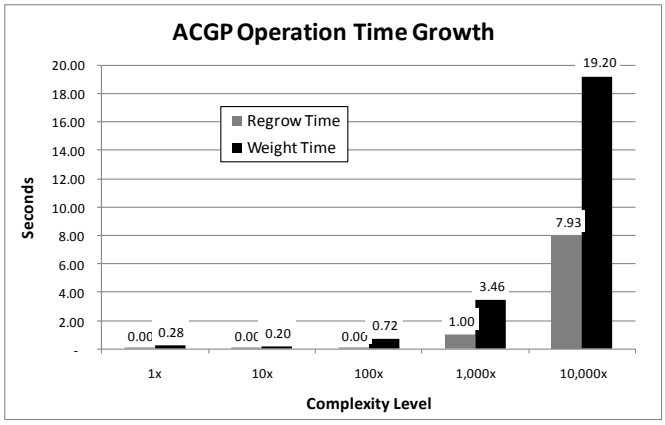

Figure 5 ACGP Regrow and Weight times from Table 3

Regrow and Weight time, the fastest growing times from Table 3, are better illustrated in Figure 5. Weight time is the most rapidly growing measure – it includes tree traversals and also processing of all heuristic tables, and with larger cases we observed memory paging coming into play causing the increases. This issue is further tested next.

Table 4 REGROW AND OPERATOR TIMES (UP TO 11 SIMULTANEOUS PROCESSES)

	Regrow	Operator
1	8.000	5.000
2	10.500	6.000
3	12.000	5.667
4	15.750	6.500
5	14.400	9.200
6	12.333	28.667
7	10.429	116.571
8	10.750	302.625
9	11.444	371.667
10	10.100	454.900
11	767.455	26880.909

The above timing does not take into account the RAM bottleneck – even if computer system supports large process space and programming language supports large indexes, when some data is flushed out of RAM to VM this will affect the overall performance. To simulate this scenario, we selected the 10,000x case, which requires close to 700k space (see Table 3), and we force a number of such processes to run concurrently, competing for RAM memory and thus causing page faults. The more page faults per timed unit, the larger the impact. In this experiment, we measure Regrow and Operator time, which both need to access the heuristics. The results are presented in Table 4 for the 1-11 concurrent processes, and illustrated graphically in Figure 6 using log scale.

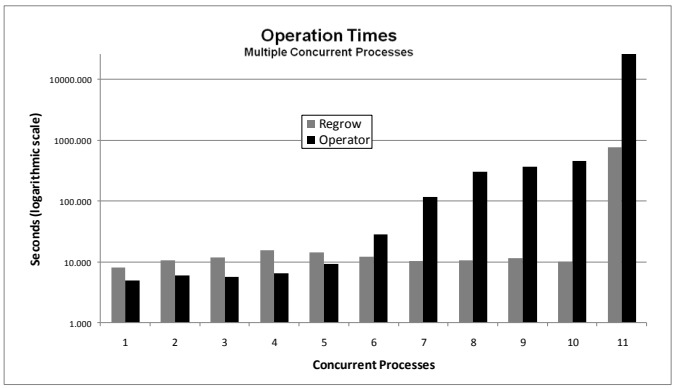

Figure 6 Regrow and Operator time for 1-11 concurrent executions of the 10,000x case, on a logarithmic scale

The growth of these times captures some of the process contention issues as the complexity grows. The Regrow time is fairly constant through 10 executions, due to kernel policy with would hardly ever interrupt during a single regrow – however, the kernel started interrupting when running 11 processes due to more demand on the system, and the time starts growing rapidly. The Operator time grows slowly when a few processes compete for RAM, but again for the same reasons shots up for the 11 processes.

Table 5 summarizes the average total processing time for the runs, showing the same slow growth for 1-10 competing processes, coming to an explosion when the demand exceed the system resources or kernel policies for the 11 simultaneous processes.

Table 5 TOTAL EXECUTION TIMES (11 SIMULTANEOUS PROCESSES)

	Total time
1	217.000
2	267.000
3	336.000
4	457.250
5	489.000
6	653.500
7	785.714
8	892.625
9	1,156.222
10	1,307.444
11	97,127.273

IV. Conclusions and Future Work

The results indicate that with larger heuristics, and a growing mismatch between available RAM and total process space, the cost of processing the heuristics will grow too fast to

possibly be offset by any of the possible benefits. From the simulations and computation, it seems that third order heuristics, in the current format and method, will be as far as ACGP could go and even that would require dedicated computer and may become prohibitive with larger number of labels or especially with ternary, or higher arity function.

When a problem exhibits some structure, a well designed mechanism can exploit this structure to improve search efficiency. When speaking of search efficiency, one has to weigh the costs of space and time versus the benefits. We have shown that for problems with strong structure, ACGP running with first or second order heuristics can exploit the structure and lead to improvements both on generation scale as well as on time scale (thus taking cost into account).

On the other hand, we have also shown that the cost of processing the heuristics would become prohibitive beyond second or third level, depending on the number of labels and on function arities. This cost is mostly due to the mismatch between available RAM and the total process space available on a given computer system. One obvious solution to allow the benefits while keeping cost low would be to process the heuristics using the current explicit mechanisms only up to the second order, and apply some machine learning techniques beyond second order to process only the most plausible heuristics. One way this could be accomplished with minimum implementation overhead would be to use so called *growing language* – in addition to the initial function and terminals labels present in a given environment, include also artificial labels referring to individual best performing first and second order heuristics – in essence allowing the heuristics to combine to form deeper order heuristics. The space requirements for this would be minimal as such new labels would be treated as terminals, which from Eq. 4 and 5 affect the space required very modestly for first order and second order ACGP.

REFERENCES

[1] Janikow, Cezary Z. and Aleshunas, John J., *Second Order Heuristics with ACGP*. unpublished.

[2] Banzhaf, W., Nordin, P., Keller, R., Francone, F., *Genetic Programming – An Introduction*, Morgan Kaufmann, 1998.

[3] Langdon, W., Poli, R., *Foundations of Genetic Programming*, Springer, 2002

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

[5] 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

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

[7] Janikow, Cezary Z. *Evolving Problem Heuristics with On-line ACGP*, GECCO'07, July 7–11, 2007

[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] 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