Using Compression to Understand the Distribution of Building Blocks in Genetic Programming Populations

McKay, R. I. (Bob), *Senior Member, IEEE*, Shin, Jungseok,
Hoang, Tuan Hao, Nguyen, Xuan Hoai *Member, IEEE* and Mori, Naoki *Member, IEEE*

***Abstract*— Compression algorithms generate a predictive model of data, using the model to reduce the number of bits required to transmit the data (in effect, transmitting only the differences from the model). As a consequence, the degree of compression achieved provides an estimate of the level of regularity in the data. Previous work has investigated the use of these estimates to understand the replication of building blocks within Genetic Programming (GP) individuals, and hence to understand how different GP algorithms promote the evolution of repeated common structure within individuals. Here, we extend this work to the population level, and use it to understand the extent of similarity between sub-structures within individuals in GP populations.**

I. Introduction

Pattern repetition in Genetic Programming (GP) populations has recently become a hot topic. While the initial emphasis was on the easier-to-analyse linear GP [11], [14], the focus has subsequently shifted to tree-based GP [10]. These studies cast considerable light on the evolution of GP populations, and the spread of structures within populations.

Parallel to this development, we are studying the evolution of repeated patterns within GP individuals, in problems where repetition of components of solution structure is desirable [13]. We found tree compression a useful tool, providing an understanding of the extent of repetition within genotypes, and allowing comparison between the spread of code repetition under different GP algorithms, both in effective and in non-effective code.

However tree compression can not only tell us about repetition of code within individuals, but, applied to whole populations, also about the spread of code between individuals. We view this as complementary to [10], providing information about different aspects of code repetition. While the latter tells us a great deal about exactly- matched subtrees, the former gives us information about the overall extent of matching, including inexact matching, being based on a probabilistic compression algorithm.

In this study, we are interested in regularity and repetition not only in the genotype as a whole, but also in the effective part: that is, whether the changes in regularity in the effective code mirror those in the whole genotype, or whether they behave in different ways. Hence this paper also emphasises use of expression simplification to yield, as far as possible, the effective skeleton of code.

We study this behaviour through two scenarios – comparisons of different GP algorithms – in which we might discern differences in the emergence, propagation and preservation of building blocks. To simplify presentation, both use the same GP problem, a symbolic regression problem in which the target solution is highly regular. We emphasise that the primary aim of this paper is to demonstrate the analyses that compression techniques support, not to make particular claims about these algorithms. We do not intend to claim general applicability of these results, but only of the techniques. Compression analyses of these GP systems on other problems might show completely different behaviour, invalidating general conclusions drawn from these results. However it would only serve to *confirm* the value of the compression analyses.

The rest of the paper is structured as follows: the remainder of this section gives a brief background to compression and simplification. Section II explains the approach, and introduces our compression-based metrics and the evolutionary scenarios we use for the investigation. Section III presents the results of these experiments, and in Section IV we examine the meaning of these results, and discuss the insight they give into the behaviour of these evolutionary systems. Finally, Section V presents our overall conclusions, discusses the assumptions and limitations of the work, and suggests directions in which the work may be extended.

A. Compression

In data compression, the aim is to encode the information from a file in fewer bits than the original format. Here, we focus on lossless compression. Most compression algorithms use a two-stage approach, in which the data is first modelled, and then encoded using the model. The compression algorithm considered in this paper uses a statistical model. It relies on the ability to detect and model statistical regularities in the data. Of course if the data does not contain such regularities, compression will fail: the new file will be larger than the original file. An underlying assumption is that regularities in our GP populations will fit the model of the particular state-of-the-art compression algorithm we use, sufficiently well that the degree of compression will reflect the degree of regularity in the data.

1) String vs Tree Compression: The compression literature has historically focussed primarily on string compression

Bob McKay and Jungseok Shin are in the CSE Dept, Seoul National University, Korea; Hao Hoang is in the School of IT & EE, University of New South Wales @ ADFA, Canberra, Australia; Xuan Nguyen is in the CS Dept, VietNam Military Technical Academy, Hanoi, VietNam; Naoki Mori is in the CSS Dept, Osaka Prefecture University, Japan

algorithms, such as the well-known Ziv-Lempel [16]. However such algorithms are ill-suited to compress population data from tree-based GP. Although tree data can be linearly represented, e.g. by an inorder traversal of the tree, this introduces spatial biases into the representation (left children are close to their parents, but other children are not). So they also introduce bias into the compression: regularities involving left children will likely be discovered, but other regularities will not, so string compression algorithms are ill-suited to our purpose. Instead, we use an XML compression system, XMLPPM [1].

XMLPPM is based on the well-known Predict by Partial Match (PPM [2]) algorithm, which like most statistical compression methods provides higher compression ratios than dictionary-based methods, at higher computational cost. PPM uses a Markov model to condition the probability that a particular symbol will occur, on the sequence of characters which immediately precede the symbol. The length of context used to predict the next symbol is determined by the order of the Markov model (see [13] for details).

XMLPPM extends this model to trees using a stack to record the context at each branch. When compression of a branch is completed, the stack is popped to recover the context at the last remaining branch point; thus the same context is used to predict all children of a given node, removing any bias in the compression model. Although XMLPPM compresses trees represented as XML documents, this is a minor technical detail; conversion of a tree to XML representation is a straightforward matter, and does not affect the degree of compression achieved.

B. Expression Simplification

In expression simplification, we aim to apply transformations to an expression, to obtain a simpler expression with the same semantics. This is important in GP, in order to obtain the effective core of the code which is being evolved.

Most research [9], [7], [4], [15] uses syntactic methods to simplify trees. We can simplify an arithmetic expression tree by applying arithmetic. For example, in an arithmetic expression tree, for any term T, $T*0$ or $0*T$ simplifies to 0. Of course, there is a wide range of mathematical identities which may be used. However, rule-based simplification cannot find all available semantic redundancies. We proposed an alternative, semantically based method, Equivalent Decision Simplification (EDS) [12], and presented evidence that, at least for this domain, EDS found far more simplifications than did rule-based simplification, and closely approximated a complete search of the available simplifications.

II. Methods

We combine the two methods, EDS and PPM-based compression of populations of trees, to obtain further insights into the behaviour of a number of different algorithms. In fact, we base the study on two previous sets of experiments, which compared a number of algorithms. Both sets of experiments compared GP systems originally designed to promote replication of building blocks within individuals.

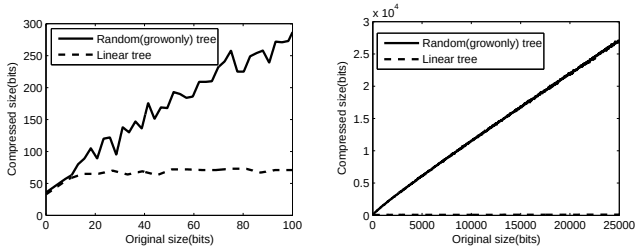

Fig. 1. Compression of Random and Linear Trees vs Tree Size (left: Small Trees; Right: Large Trees)

With the exception of the Koza-style GP, all algorithms were based on the TAG representation detailed in [5]. We summarise some key properties:

1) TAG representation is based on labelled trees, as in Koza-style GP. using subtree crossover and mutation
2) Any rooted subtree of a TAG tree, or any consistent extension, is valid: deleting any subtree of a TAG tree yields a TAG tree, and a TAG tree may be extended by adding any valid subtree at the leaves
3) While TAG-based algorithms evolve a TAG representation, this representation is converted to a standard Koza-style expression tree prior to evaluation

Because of the validity properties, it is possible to use TAG in many ways. In this paper, we consider two such applications:

- Adding code-duplicating local search to TAG3P
- A TAG-based developmental evaluation system

A. Compression Issues

While statistical compression algorithms compress large, structured objects well, they have overheads which lead to difficulty in compressing small or very randomly structured objects. Thus the absolute compression ratio can be misleading. For very unstructured objects, it may be greater than one, while for small objects, we may get poor compression even if the object is highly structured. This means that it is difficult, using raw compression ratios, to know the overall scale of the compression space, or to compare compression ratios between objects of very different sizes.

To overcome this, we take a further step. With PPM, random structures will compress worse than any other structures. On the other hand, linear trees, with only a single internal symbol, will compress near-optimally. We use these extremes as normalisation bounds on the compression ratios. In detail, we generated a large number of near-random trees (using the 'grow' part of the initialisation process for GP populations)[1]. We also generated linear trees of all possible sizes within the range of interest. We compressed these individuals with

[1]In [13], we used both full and grow trees. With the former, XMLPPM can use a reduced character set for arithmetic coding of internal nodes, and hence get better compression: full trees are significantly non-random. We erred in our view "Note...the step in compression of random trees...probably an artefact of the fixed-size context used by PPM algorithms.". This, plus our use of elitism, altered the three figures from [13] slightly, but the conclusions of that paper are unaffected.

TABLE I

COMPRESSION AND SIMPLIFICATION PARAMETER SETTINGS

PPM Context Depth	16
Method	EDS
Simplification Relative Error	10^{-4}
Subtree search order	Breadth-first from leaves
Candidates for substitution	$(0),(1),(x)$

TABLE II

GRAMMAR DESCRIBING SOLUTION SPACE

$G = V,T,P,S$	$EXP \rightarrow EXP\,OP\,EXP$
$S = EXP$	$\mid sin\,EXP \mid VAR$
$V = EXP,OP,VAR$	$OP \rightarrow + \mid - \mid * \mid /$
$T = x,sin,+,-,*,/$	$VAR \rightarrow x$

XMLPPM. The results are plotted in figure 1[2].

We interpolated this graph to obtain lookup tables for the normalisation range. In detail, if a size S tree compresses to size C, we find in the normalisation table the extreme compression values L and R for trees of this size, and compute the ratio $(R-C)/(R-L)$. This metric has some desirable properties:

- For trees of a given size, it is monotonic with compression (and thus estimates the degree of regularity)
- It normalises the regularity between 0 and 1 (random trees – with no regularity – will give a value of 0, while linear trees – with close-to-maximal regularity from the perspective of XMLPPM – will give a value of 1)
- It discounts any compression algorithm overheads

While the genotype representations considered here differ, all are eventually converted to a standard expression tree for evaluation. For comparability, all compression measurements are performed on expression trees, not on the primary genotypic representation. Compression and simplification parameter settings are given in table I.

B. Problem Domain

The problem domain was a symbolic regression, in which GP was tasked to find a function fitting 20 points generated by a simple polynomial expression – in this case, $F_n(x) = x + x^2 + x^3 + ... + x^n$, for various values of n. We used the grammar in table II to define the solution space. After EDS, further symbols may appear: V is extended with a further non-terminal $CONST$, and T with two further terminals, $0,1$, while P is extended with two additional productions: $EXP \rightarrow CONST$ and $CONST \rightarrow 0|1$. For our first analysis, we drew on experiments from [5] using the duplication operator, which selects a random node in the current tree, copies the subtree rooted at that point, and re-inserts it at another point in the tree. Since the duplication operator alone results in uncontrolled bloat, we balanced it with a truncation operator, which deletes a randomly-selected subtree. We anticipated that duplication would be very effective in such a highly structured problem domain. In fact, it performed relatively poorly as a mutation operator, but worked well as a local hillclimbing search operator when combined with crossover as an evolutionary operator. Further details of the experiments are given in [5]. To summarise, we compared five versions: TAG3P which uses subtree crossover and mutation; TAGCROSS, which omits mutation entirely; TAG3PM, which substitutes subtree mutation with balanced duplication and truncation; and LSTAG3P10 and LSTAG3P50, which omit mutation, but use respectively 10 and 50 steps of local search with balanced duplication and truncation. These experiments used the function F_9. Detailed parameter settings are given in tables III and IV.

[2]Where more than one random tree of a given size was found, we used the worst compression ratio).

TABLE III

COMMON EXPERIMENTAL PARAMETER SETTINGS

Elitism	Elite of 1
Terminal Operands	X (the independent variable)
Operators (Function set)	$+,-,*,/,sin$
Fitness Cases	The sample of 20 points in the interval $[-1..+1]$
Fitness	sum of errors over 20 fitness cases
Runs per experiment	30

TABLE IV

DUPLICATION EXPERIMENT: PARAMETER SETTINGS

Objective	Find a function fitting a given sample of 20 (x_i,y_i) data points, for target function F_9
Hits	The number of fitness cases for which the error is less than 0.01
Variation Operators	subtree crossover and subtree mutation for both GP and TAG3P
Common Parameters	$MAX_{GEN} = 51, MAX_{SIZE} = 40$, Crossover rate=0.9, mutation rate=0.1
Evolutionary runs	$POP_{SIZE} = 500$
Local search 10	$SEARCH_{DEPTH} = 10, POP_{SIZE} = 50$
Local search 50	$SEARCH_{DEPTH} = 50, POP_{SIZE} = 10$
Success predicate	An individual scores 20 hits

For the second analysis, we drew on experiments on developmental evaluation [6], which has been proposed as a mechanism to promote structural regularity in GP. In analogy with biological systems, this mechanism evaluates individuals throughout development to promote adaptibility, and hence select for regular structure. For developmental evaluation, it is essential to have a series of problems of increasing difficulty; in this case, we used the function set $F_1,...F_9$. We simulate developmental evaluation by evaluating the individuals, as they grow, on successively more difficult problems from the problem family. In these experiments, five treatments were used: standard Koza-style GP, the TAG3P system as above; DEVTAG, which uses TAG3P's evolutionary algorithm, but replaces the evaluation of a single objective with evaluation during 'development' (which is taken to mean, sub-sections of the TAG tree); DTAG3P, a typical developmental system, evolving an L-system which generates an individual through a series of stages, evaluated during development as above; and DTAG3PF9ALL, which uses the DTAG3P developmental process, but conducts all evaluations on the final function, F_9 (as do standard GP and TAG3P). Detailed parameter settings are given in tables III

TABLE V

DEVELOPMENTAL EXPERIMENT: PARAMETER SETTINGS

Objective	Find a F_9 (GP, TAG3P) or $F_1, F_2, \ldots F9$ (DEVTAG, DTAG3P) fitting a sample of 20 (x_i, y_i) data points
Success Predicate	Sum of errors $< \varepsilon = 0.01$
Variation Operators	crossover between rules, sub-tree crossover and sub-tree mutation on successors for DTAG3P; sub-tree crossover and sub-tree mutations for GP, TAG3P and DEVTAG
Probabilities	Crossover 0.9; mutation 0.1
TAG Min/Max initial size	2 to 1000
Max depth for GP	20
# of DTAG3P rules	8
Min/Max # of β-trees in each successor (DTAG3P)	1 to 5
Population size	250

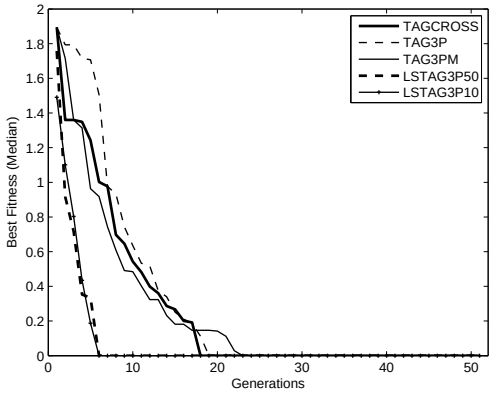

Fig. 2. Duplication Experiment: Median Best Fitness vs Generation

and V.

III. RESULTS

Figure 2 shows the fitness of the best individual in each generation, averaged over 30 runs. Figure 3[1] shows the structural regularity for the fittest individual in each run, averaged over the 30 runs. Figures 4, 5 were constructed by taking the best individual in a generation from each of the 30 runs, and compressing the whole. Figures 6, 7 show the results of compressing the whole population in each generation (again, averaged over 30 trials). Table VI shows

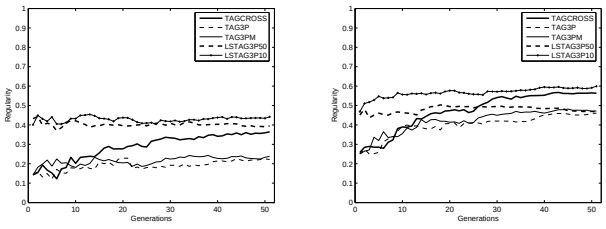

Fig. 3. Duplication Experiment: Individual Complexity vs Generation (left: Unsimplified Trees; right: Simplified Trees)

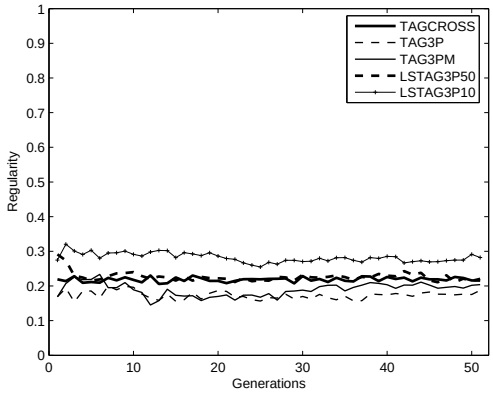

Fig. 4. Duplication Experiment: Between-Experiments Complexity: Unsimplified Trees

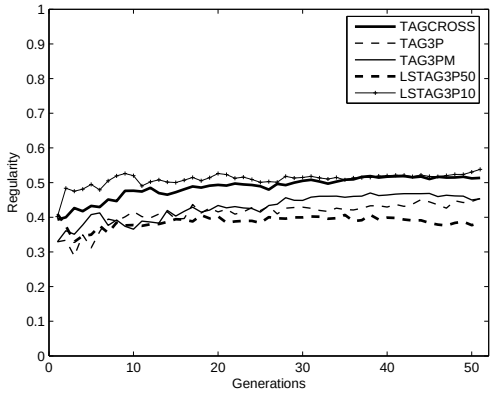

Fig. 5. Duplication Experiment: Between-Experiments Complexity: Simplified Trees

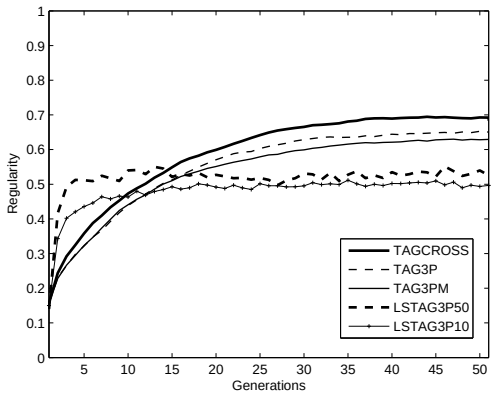

Fig. 6. Duplication Experiment: Population Complexity: Unsimplified Trees

TABLE VI

DUPLICATION EXPERIMENT: END OF RUN COMPLEXITY VALUES[3]

INDIVIDUAL MEANS INDIVIDUAL COMPLEXITY; POPULATION MEANS POPULATION COMPLEXITY; BETWEEN MEANS BEWEEN-RUN COMPLEXITY; RAW MEANS UNSIMPLIFIED TREES; SIMP. MEANS SIMPLIFIED TREES

Complexity Measure	SYSTEM	TAGCROSS	TAG3P	TAG3PM	LSTAG3P50	LSTAG3P10
Individual	Raw	0.3647 ± 0.1452	0.2220 ± 0.1187	0.2383 ± 0.1415	0.3949 ± 0.1285	0.4416 ± 0.1772
	Simp.	0.5634 ± 0.1674	0.4605 ± 0.1545	0.4718 ± 0.1689	0.4689 ± 0.1821	0.5994 ± 0.1717
Population	Raw	0.6921 ± 0.0335	0.6504 ± 0.0219	0.6291 ± 0.0227	0.5256 ± 0.0668	0.4967 ± 0.0370
	Simp.	0.8090 ± 0.0455	0.7710 ± 0.0401	0.7424 ± 0.0401	0.5691 ± 0.0899	0.5997 ± 0.0355
Between	Raw	0.2157	0.1875	0.2038	0.2204	0.2822
	Simp.	0.5134	0.4542	0.4535	0.3841	0.5383

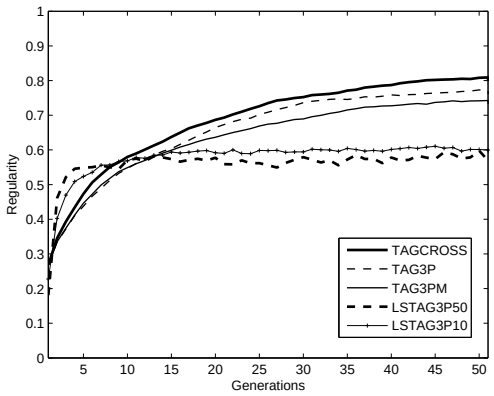

Fig. 7. Duplication Experiment: Population Complexity: Simplified Trees

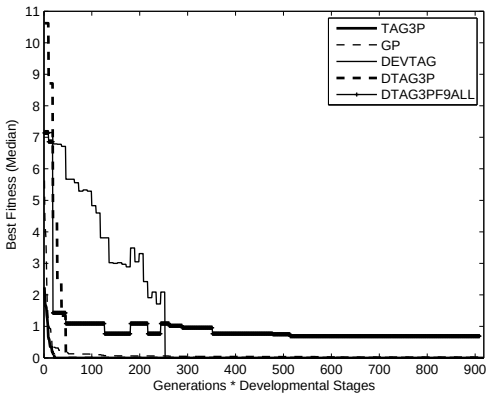

Fig. 8. Developmental Experiment: Median Best Fitness vs Time[4]

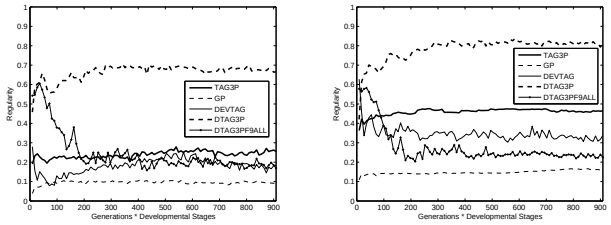

Fig. 9. Developmental Experiment: Individual Complexity vs Time[4] (left: Unsimplified Trees; right: Simplified Trees)

the end-of-run values for these measurements[3].

Figures 2 and 3 are provided primarily for reference, since they are discussed in more detail in [13]. Points to note in the latter include the high regularity generated by local search with duplication/truncation, especially in the effective code; the gradual increase in regularity when crossover is the only operator, i.e. in the absence of any form of mutation, again especially for effective code; and the reduced regularity in effective code generated with a greater depth of local search.

Figures 4 and 5 may be seen as estimating the extent of regularity between runs – the propensity of the particular system to construct its best solutions from the same components in repeated runs. The most notable point is the much higher regularity in effective code than in overall code, and we note a trend toward increasingly regular effective code in the crossover-only runs, shared more weakly by the runs in which duplication/truncation functioned as mutation operators.

In figures 6 and 7, the two local search treatments behave very similarly in population regularity, as do the other three, all eventually reaching much higher regularity, with the crossover-only treatments generating slightly lower regularity than the other two. The effective code exhibits substantially more regularity than the ineffective code.

In the developmental evaluation experiments, the figures follow the same structure as for the duplication experiments[4].

Figure 8 shows the fitness of the best individual in each generation, averaged over 30 runs. Figure 9[1] shows the structural regularity for the fittest individual in each run, averaged over the 30 runs. Figures 10 and 11 were constructed by taking the best individual in a generation from

[3]Individual complexity is measured as the mean and standard deviation (over all runs) of the compression complexity of the best individual in the run; population complexity is measured as the mean (over all runs) of the compression complexity of the whole population. Between-runs complexity is measured as the compression complexity of the set formed from the best individual of each of the runs, so no mean or standard deviation is calculated.

[4]In the developmental figures, the X axis shows the number of generations times the number of developmental stages, so as to give a fair comparison – in terms of function evaluations – between developmental and non-developmental systems.

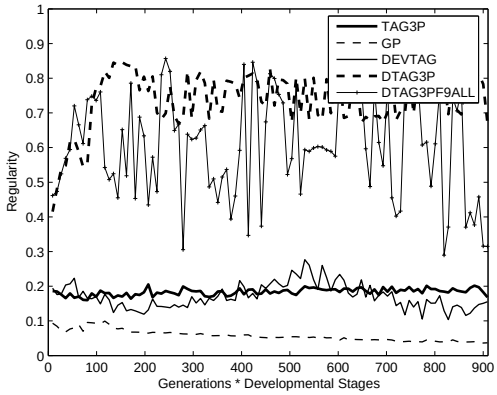

Fig. 10. Developmental Experiment: Between-Experiments Complexity vs Time[4]: Unsimplified Trees

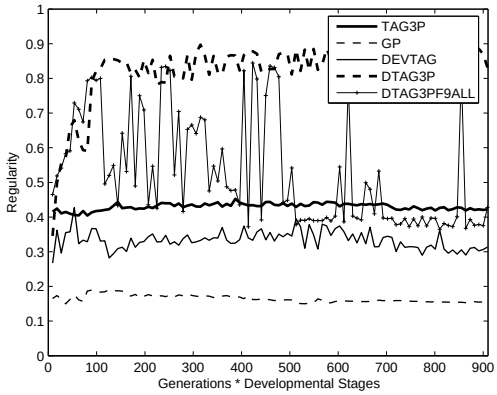

Fig. 11. Developmental Experiment: Between-Experiments Complexity vs Time[4]: Simplified Trees

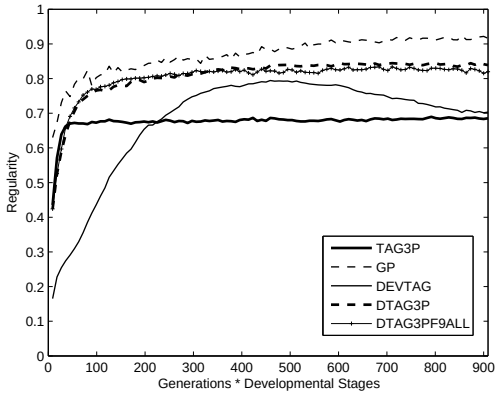

Fig. 12. Developmental Experiment: Population Complexity vs Time[4]: Unsimplified Trees

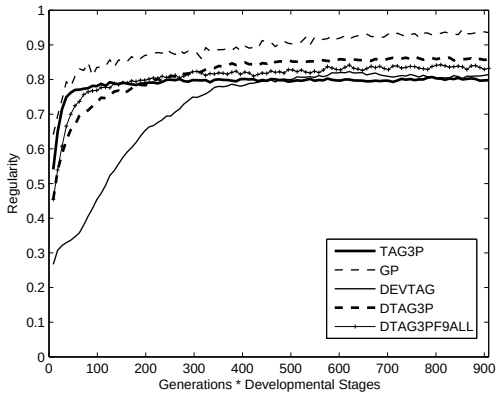

Fig. 13. Developmental Experiment: Population Complexity vs Time[4]: Simplified Trees

each of the 30 runs, and compressing the whole. Figures 12 and 13 show the results of compressing the whole population in each generation (again, averaged over 30 trials). Table VII shows the end-of-run values for these measurements.

Figure 9 reveals that developmental evaluation with an L-system developmental process generates by far the highest regularity. If developmental evaluation is omitted, an L-system can initially generate high regularity, but it is rapidly lost. Regularity is generally higher in the effective code than in the ineffective. Standard GP exhibits substantially lower regularity than any of the other systems, especially in effective code.

Interestingly, figures 10, 11 show a large difference between the uniformity of building blocks found by different runs with L-systems development, and with non-developmental evolution. In the latter case, there is very little evolution of regularity in the overall code, and not much more in the case of standard GP in the effective code. That is, different runs do not discover similar building blocks to any great extent. For TAG-based runs, there is a little more regularity in the effective code components, but it is only modest. By contrast, the L-system developmental systems discover very similar solution components in every run, especially in the case of DTAG3P.

At the population level, figures 12, 13 show intriguing effects. All three developmental systems show greater evolution of overall regularity (that is, spread of building blocks) than does the non-developmental TAG system. Surprisingly, this effect disappears in the effective code. All systems exhibit a (high) degree of regularity. On the other hand, standard GP shows higher regularity still, in both overall and effective code, than do any of the other systems.

IV. Discussion

A. Duplication/Truncation Experiments

Compression between runs tells us about the propensity of runs to discover solutions composed of similar components. From these figures, we can see that effective code exhibits

TABLE VII

DEVELOPMENTAL EXPERIMENT: END OF RUN COMPLEXITY VALUES[3]

INDIVIDUAL MEANS INDIVIDUAL COMPLEXITY; POPULATION MEANS POPULATION COMPLEXITY; BETWEEN MEANS BEWEEN-RUN COMPLEXITY; RAW MEANS UNSIMPLIFIED TREES; SIMP. MEANS SIMPLIFIED TREES

Complexity Measure	SYSTEM	TAG3P	GP	DEVTAG	DTAG3P	DTAG3PF9ALL
Individual	Raw	0.1672 ± 0.1267	0.0967 ± 0.1001	0.1707 ± 0.0946	0.7125 ± 0.2420	0.4143 ± 0.2660
	Simp.	0.2174 ± 0.1144	0.1795 ± 0.1027	0.3269 ± 0.2226	0.6535 ± 0.3083	0.5563 ± 0.3436
Population	Raw	0.6772 ± 0.0457	0.9340 ± 0.0545	0.7672 ± 0.1044	0.8276 ± 0.0164	0.8377 ± 0.0433
	Simp.	0.8156 ± 0.0227	0.9650 ± 0.0342	0.8454 ± 0.0736	0.8468 ± 0.0210	0.8492 ± 0.0509
Between	Raw	0.2417	0.0818	0.1942	0.8851	0.8841
	Simp.	0.3461	0.1673	0.2796	0.8683	0.9038

far more regularity than ineffective code, yet the within-individuals regularity is generally similar for overall and for effective code. We thus reach the (not altogether surprising) conclusion that, at least for this problem, selection pressure has acted to ensure that the effective code finds similar building blocks in each run, but that this effect is weaker, if present at all, for the ineffective code.

It is difficult to determine whether the increase in regularity seen in between-runs compression indicates the gradual discovery of similar building blocks, or simply reflects the increase of regularity within the individual solutions. However the effect is most marked in those cases – crossover-only and duplication/truncation-mutation-only – where an increase in within-individual is most marked, suggesting that the latter is quite likely.

It is worth noting that this inability to separate such effects constitutes one of the main limitation of these compression techniques. While regularity effects might be observed, it is often difficult to determine precisely what is the source of the regularity, since there is no sensible way to compress a set of trees without also compressing the individual trees.

From the population compression results, we can draw a number of interesting conclusions. First, it is clear that the effective code has a much higher tendency to discover common building blocks than does the ineffective code, presumably because of the higher selective pressure it is subject to. Second, that the selection of such components continues throughout the run, particularly for the effective code. Third, that the effect is stronger in the case of crossover-only runs, perhaps because mutation exerts a slight disruptive effect on the spread of building blocks. Fourth, that there is a much lower rate of increase of regularity in the local search runs; this is partially explainable because the runs are calibrated to use the same number of evaluations. Since local search runs are evaluated in every step of local search, there are correspondingly fewer opportunities for crossover to spread code components between individuals. This would partially explain the lower rate of increase of regularity, but the explanation is incomplete, since it does not explain why the two local search runs have such similar rates of increase despite the fivefold disparity in their crossover rate.

B. Developmental Experiments

The results on individual regularity mostly fitted our expectations, that developmental methods could generate regularity, but that without developmental evaluation, they could not select for it in effective code.

With between-runs regularity, the complete dichotomy between developmental and non-developmental systems is noteworthy. Developmental runs clearly generate the same components over and over in different runs, but non-developmental systems do not. This similarity is not simply the result of internal self-similarity, since it occurs with DTAGF9ALL, which shows no tendency toward evolving self-similarity.

The population results were quite surprising, especially the very high level of regularity generated by standard GP. That is, standard GP is much more effective at spreading code components throughout the population than any of the TAG-based approaches - even TAG3P, whose evolutionary process is identical with that of standard GP. This is not inconceivable, since even if TAG3P exhibits a similarly high propensity to generate similar building blocks in its native representation, it is possible that the 2-stage mapping which generates the expression tree representation from the original TAG representation may distort such regularity. But it is surprising, since one might expect such distortion to also result in the TAG-based systems performing poorly – the reverse of our experience.

V. CONCLUSIONS

A. Implications of the Work

The main conclusion of this work is that compression-based metrics can tell us a great deal about the behaviour of GP systems, and in particular, about their tendency to share components across a population, or to find similar components in different runs. We were able, in section IV, to draw surprisingly detailed conclusions about the behaviours of the different algorithms, at least on the particular problem studied here. In this page-limited paper, we chose to investigate the range of questions about building blocks and code replication that could be handled by compression analyses, rather than extending to different problems, since results on different problems would be of equal value whether they

were confirmatory of the results obtained here, contradictory to them, or neutral. All would be interesting.

B. Assumptions and Limitations

The PPM tree compression model assumes that a node is predicted well by its context of parent nodes. Thus the degree of compression reflects the extent to which this model applies to some data. That is, high levels of compression imply that it is often the case that the same parent nodes are often followed by the same child nodes. It is our contention that this corresponds closely to the normal understanding of GP building blocks.

Our work is also based on the assumption that linear interpolation between the extremes of compressibility and incompressibility is a reasonable way to estimate regularity; and on the twin assumptions that linear and random trees lie at those extremes. Everything we have observed so far is consistent with these assumptions.

Of course, in undertaking this work, we are comparing regularities of trees of vastly different sizes, especially in the developmental experiments [8]. Does this introduce a bias (independent of the previous issue)? This question introduces deep issues of the meaning of complexity, far beyond the scope of this paper; however we note that complexity definitions based on information theory and compression form one of the primary strands of research in defining complexity [3].

The primary limitation of this approach lies in our inability to use it to answer some natural questions about the source of regularity. For example, we can estimate the regularity of GP populations, and the regularity within individuals. However we have no way to directly measure the difference. We can't specify a metric for the degree of repetition between individuals, since we can see no way to remove the effect of the within-individual regularity.

In principle, the approach is applicable to any GP tree representation since it can be converted to XML format. However in the absence of a standardised GP tree representation, this mapping has to be generated for each particular representation. For this analysis (and it must be said, for other GP analyses also), it would be highly desirable to have a standard XML representation for tree-based GP populations.

C. Future Extensions

In its present form, the compression analysis can only estimate the extent to which GP components are repeated across a population; it gives no indication of how those components are conserved from generation to generation. In principle, however, compression approaches could achieve this. If we were to use the population from one generation, generation t, to generate a compression model, but use that model to compress a second generation, generation $t+k$, we would in effect be estimating the extent to which common components from the first population were repeated in the second. That is, we would have a way to measure the conservation of components from generation to generation. We are currently working on this extension.

ACKNOWLEDGMENT

We thank James Cheney for assistance with XMLPPM and Moonyoung Kang for building the EDS scripts. We thank them, and Daryl Essam, for insights that led to the present paper. This work was supported by the Research Settlement Fund for new faculty of Seoul National University.

REFERENCES

[1] J. Cheney. Compressing xml with multiplexed hierarchical models. In *IEEE Data Compression Conference*, pages 163–172, Snowbird, Utah, 2002.

[2] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32:396–402, 1984.

[3] B. Edmonds. Syntactic measures of complexity, 1999.

[4] A. Ekart. Shorter fitness preserving genetic programs. In C. Fonlupt, J.-K. Hao, E. Lutton, E. Ronald, and M. Schoenauer, editors, *Artificial Evolution. 4th European Conference, AE'99, Selected Papers*, volume 1829 of *LNCS*, pages 73–83, Dunkerque, France, 3-5 Nov. 2000.

[5] N. X. Hoai, R. I. McKay, D. Essam, and H. T. Hao. Genetic transposition in tree-adjoining grammar guided genetic programming: The duplication operator. In M. Keijzer, A. Tettamanzi, P. Collet, J. I. van Hemert, and M. Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 108–119, Lausanne, Switzerland, 30 Mar. - 1 Apr. 2005. Springer.

[6] T. Hoang, D. Essam, R. McKay, and X. Nguyen. Developmental evaluation in genetic programming: A tag-based framework. In *Proceedings of the third Asia-Pacific Workshop on Genetic Programming*, VietNam Military Technical Academy, Hanoi, VietNam, Oct. 12-14 2006.

[7] D. Hooper and N. S. Flann. Improving the accuracy and robustness of genetic programming through expression simplification. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, page 428, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[8] M. Kang, J. Shin, T. Hoang, R. McKay, D. Essam, N. Mori, and X. Nguyen. Code duplication and developmental evaluation in genetic programming. In 10^th *Asia-Pacific Workshop on Intelligent and Evolutionary Systems*, pages 181–191, Seoul, Korea, 2006.

[9] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[10] W. B. Langdon and W. Banzhaf. Repeated patterns in tree genetic programming. In M. Keijzer, A. Tettamanzi, P. Collet, J. I. van Hemert, and M. Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 190–202, Lausanne, Switzerland, 30 Mar. - 1 Apr. 2005. Springer.

[11] W. B. Langdon and W. Banzhaf. Repeated sequences in linear genetic programming genomes. *Complex Systems*, 15(4):285–306, 2005.

[12] N. Mori, R. McKay, X. Nguyen, and D. Essam. How different are genetic programs: New methods for studying diversity and complexity in genetic programming. *in review*, 2007.

[13] J. Shin, M. Kang, R. McKay, X. Nguyen, T. Hoang, N. Mori, and D. Essam. Analysing the regularity of genomes using compression and expression simplification. In *Genetic Programming 10th European Conference, EuroGP 2007, Proceedings, Springer LNCS*, volume 4445, pages 251–260, Valencia, Spain, 2007.

[14] G. C. Wilson and M. I. Heywood. Context-based repeated sequences in linear genetic programming. In M. Keijzer, A. Tettamanzi, P. Collet, J. I. van Hemert, and M. Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 240–249, Lausanne, Switzerland, 30 Mar. - 1 Apr. 2005. Springer.

[15] P. Wong and M. Zhang. Algebraic simplification of genetic programs during evolution. Technical Report CS-TR-06-7, Computer Science, Victoria University of Wellington, New Zealand, 2006.

[16] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978.