Code Duplication and Developmental Evaluation in Genetic Programming

Kang, Moonyoung[1], Shin, Jungseok[2], Hoang, Tuan Hao[3], McKay, RI (Bob)[4], Essam, Daryl[5], Mori, Naoki[6], and Nguyen, Xuan Hoai[7]

[1] Seoul National University, Seoul, Korea `yerihyo@hotmail.com`
[2] Seoul National University, Seoul, Korea `shin119@yahoo.com`
[3] University of New South Wales @ ADFA, Canberra, Australia `haohth@hotmail.com`
[4] Seoul National University, Seoul, Korea `rim@cse.snu.ac.kr`
[5] University of New South Wales @ ADFA, Canberra, Australia `daryl@itee.adfa.edu.au`
[6] Osaka Prefecture University, Osaka, Japan `mori@cs.osakafu-u.ac.jp`
[7] VietNam Military Technical Academy, Hanoi, VietNam `nxhoai@gmail.com`

Abstract: We investigate a hypothesis, that structured, replicated code can be promoted by evaluation during development, and that this is the cause of the good performance of algorithms using developmental evaluation. We use compression as a tool to measure replication of code in this research. Our results show that evaluation during development does not promote replicated structured code. Hence we are left with two problems, explaining why developmental evaluation systems exhibit good performance, and understanding how replicated, structured coding has arisen in the genotype of natural biological systems.

1 Introduction

Genetic Programming (GP – [Cramer, 1985, Koza, 1992]) has been highly successful in generating solutions to difficult problems [Koza, 1994]. However GP is notorious for also generating enormously complex solutions [Banzhaf et al., 1998], with the vast bulk of the complexity being unproductive. In this respect, GP systems resemble biological evolutionary systems, in that the vast bulk of natural DNA is also unproductive. It neither encodes for proteins, nor regulates the genes that do encode proteins, and indeed is not subject to selective pressure, implying that it has no effect on the phenotype.

However the effective part of the DNA in biological evolutionary systems exhibits a high level of regularity [Schlosser & Wagner, 2004, Carroll, 2005], both in the genes themselves, and in the regulatory regions that control the expression of these genes.

At this point, we should mention an issue of terminology. Discussions of regularity, in both the biological literature and the GP literature frequently discuss the issue of modularity. It is important to recognise that the two literatures are using the term 'modularity' in different ways. In the biological literature, 'modularity' in this context generally means the grouping of a number of genes and their regulatory mechanisms into a structure which is re-used (generally by being replicated) within the genome. In the GP literature, 'modularity' generally also has a connotation, imported from Computer Science, of encapsulation and direct re-use (i.e. through calling, rather than through replication). Thus what biologists would regard as the highly modular structure of mammalian homeobox genes, replicated repeatedly with small changes in the mammalian genome, would generally be regarded as replicated, but not modular, structure in the Computer Science sense. Since the GP literature has its roots both in biology and in Computer Science, we eschew the use of the term 'modular' in this paper.

A number of mechanisms have been proposed in the GP literature to generate regularity, perhaps the best-known being Automatically Defined Functions (ADF [Koza, 1994] However it is noteworthy that these mechanisms are pragmatically motivated, and do not appear to have a direct analogue in the biological domain. Conversely, biological systems have been able to generate regularity *without* any requirement for artificial mechanisms. Moreover, mechanisms such as ADF require an a priori decision about the level of regularity required for solving a particular problem (i.e. the number of function branches to be allocated), while biological systems do not impose such a requirement. Moreover, the biological systems scale well to high-complexity problems - as witness the sheer number of homeobox genes in mammalian genomes; on the other hand, it is not clear that ADFs and similar mechanisms could scale to the same degree.

The question naturally arises, why does regularity naturally arise in biological evolutionary systems, but not in GP systems? One possibility is the developmental process which biological systems undergo, but as we have previously argued [McKay et al., 2006], this is probably not enough - while a developmental process per se *permits* the representation of regular structure in the phenotype, it is difficult to see how it could lead to selection for regularity, hence regularity will only arise by chance. We argued that evaluation during development would be required to provide a selective advantage through adaptibility to regular structures, and hence that regularity would spontaneously emerge in developmental systems incorporating evaluation during the developmental process.

Based on these ideas, our pilot study [ibid] introduced a system, DEVTAG, with a trivial developmental process (analogous to that of an undifferentiated colony animal such as a sponge), but incorporating incremental evaluation (that is, the individual is evaluated on increasingly difficult problems as it grows). We were able to demonstrate, with this system, a dramatic improvement in problem solving ability on some regular families of problems, compared both with non-developmental GP, and with a version of the system incorporating the trivial developmental process, but without developmental evaluation. We hypothesised that the improved performance was a result of increased phenotypic regularity, but at that time had no way to measure this possibility. More recently, we introduced a fully-developmental system, DTAG3P [Hoang et al., 2006a], and demonstrated an even greater improvement in performance on one family of problems. However the question remained, was our hypothesis, that the improvement was due to increased regularity induced by the developmental evaluation, correct, or was the improvement due to some other cause? The work reported here represents an attempt to answer this question, using tree compression to provide metrics.

Compression algorithms rely on detecting regularity as the means of compressing data. In particular, tree compression algoriithms rely on detecting repeated regularities in tree structures to attain compression. Hence the compression achieved by tree compression algirthms can serve as a proxy for measuring the degree of regularity in GP trees. Our aim in this paper is to measure the compression as a means of measuring regularity (higher compression means a greater degree of regularity), and hence to evaluate our hypothesis regarding the source of the good performance of our developmental evaluation systems.

The remainder of this section briefly introduces the various systems compared in these experiments - GP, TAG3P [Hoai et al., 2006], DEVTAG [McKay et al., 2006] and DTAG3P [Hoang et al., 2006a] and provides background on the simplification of arithmetic expressions, and on tree compression. Section 2 introduces the experiments and analysis conducted, while section 3 presents the results of those experiments. We discuss the results of the experiments in section 4, and present our conclusions and proposals for further work in section 5.

1.1 Genetic Programming Systems

'Standard' Genetic Programming

The GP system used in these experiments was a stock-standard Koza-style tree-based GP system, using tournament selection, and subtree crossover and mutation.

TAG Grammars and TAG3P

Tree Adjoining (TAG) Grammars were developed by Joshi [Joshi et al., 1975] as an alternative to Chomski grammars. They have particular advantages in natural language processing because of their ability to naturally represent the relationship between a simple sentence such as 'The cat sat on the mat', and more complex derived sentences such as 'The big, black cat sat quietly but menacingly on the comfortable mat which it had commandeered'. Their primary properties from a GP perspective, however, are twofold:

1. Their power: TAG grammars properly subsume context free (CFG) grammars), and any CFG may be automatically converted to a TAG
2. A feasibility property: removal of a leaf node from a valid TAG derivation tree always results in a valid TAG derivation tree

Building on the TAG representation, a TAG-based GP system, TAG3P, has previously been implemented and evaluated on a wide range of problems. The basic form of the system uses TAG derivation trees as its representation, with tournament selection and subtree crossover and mutation as its variation operators. The interested reader is referred to our previous publication [Hoai et al., 2006] for full details of the system.

Incremental Evaluation and DEVTAG

The feasibility property of TAG was crucial to the simple implementation of a prototype system as a preliminary test of our hypothesis regarding developmental evaluation. Since any rooted subtree of a TAG derivation tree is also a valid TAG derivation tree (and hence may be fitness-evaluated), it was relatively simple to adapt TAG3P to an incremental evaluation system, by evaluating increasingly large fixed subparts of each individual against increasingly difficult problems. It is somewhat debatable whether such a system is truly a developmental system (in implementation, it was not; but from a time-free perspective, it is equivalent to a system in which the individual grows in an undifferentiated way). However this is a matter of semantics. The more important point is, it provided a way to test whether this evaluation strategy could lead to improved results – which in fact, it did [McKay et al., 2006, Hoang et al., 2006b]. At the time, we hypothesised, but could not show, that the improved results reflected repeated use of the same code segments.

Developmental L-Systems and DTAG3P

A number of previous authors have used Lindenmayr (L) Systems [Lindenmayer, 1968] for developmental GP [Jacob, 1994, Haddow et al., 2001, Hornby, 2003], as a natural mechanism for representing the programmed development of an individual phenotype from its genotype. Our TAG-based L-system representation uses a simple L-system to define rewrite rules for generating TAG derivation trees; as such, it is similar to other L-system phenotype-generation mechanisms. In DTAG3P, the genotype representation is a a fixed-size set of rewrite rules; the selection operator used is tournament selection, and the variation operators are two *crossover* operators: rule exchange (in which whole rules are exchanged between children), and right-hand-side exchange (in which the right hand sides of rules may be interchanged), which may then be *followed* by a normal subtree crossover or mutation of the right hand sides of those altered rules. Our results showed that DTAG3P generated a further improvement in performance when compared with DEVTAG, and we hypothesised that this resulted from the L-system providing a more effective system to represent code replication, so that the selective bias introduced by developmental evaluation could act more effectively. The interested reader is referred to the above paper for fuller details of the experiments conducted.

1.2 Incremental Evaluation

Our incremental evaluation process uses a series of problems, of increasing difficulty. On the biological analogy that fitness at later developmental stages is only relevant if the individual survives earlier stages, we took advantage of tournament selection's important property, that it only requires a rank ordering of individuals, not a numeric fitness value. Hence we set an evaluation difference threshold ϵ. Individuals were evaluated at an early developmental stage on the first problem. Only if they gave equal performance on this first stage (within ϵ) would they go on to the next stage, otherwise the fitter individual at this stage would be selected. This process was repeated at each stage.

1.3 Effective Code

It is well-known that genetic programming leads to redundant code [Blickle & Thiele, 1994]. Such redundant parts may be replaced by smaller trees which give the same evaluation. These redundant parts can change the compression ratio of the tree, since they can appear in a tree repeatedly, while causing no difference in evaluation. Therefore, to compute the regularity properties of the effective code exactly, eliminating these unnecessary redundant parts is required.

Simplification is the process of converting a tree into an equivalent but smaller tree. The following section gives a brief introduction to tree simplification methods, including the method which was used in the following experiments.

Rule-based Simplification

We can simplify a tree by applying simple arithmetic tautologies to tree-formed expressions. For example, in an arithmetic language tree with variable set $+, -, *, /$ and terminal set $0, 1, x$, 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 is unable to simplify all available semantic redundancies (that is, finding the simplest formula corresponding to a given formula is undecidable). This is not just a practical point. For example, if the subtreee T is complex, any algorithm simplifying from the leaves inward (the most natural algorithm) will find it difficult to simplify expressions such as $(T - 1) + (1 - T)$

Equivalent Decision Simplification

A stronger simplification method, 'Equivalent Decision Simplification', has recently been proposed by Mori [Mori et al., 2007]. In Equivalent Decision Simplification, a tree and its putative simplification are evaluated over a range of inputs (generally, the inputs used to define the function to be approximated). When the values of the two trees are within a predefined error bound, the two trees are judged equivalent.

However, the problem of finding an appropriate tree for substitution remains. Making single terminal node trees candidates for substitution solve this problem. By recursively simplifying each subtree of a tree, the whole tree becomes simplified. Equivalent Decision Simplification is used to simplify trees in this experiment, so that we can investigate the behaviour of the effective code.

1.4 Compression

Compression Background

Data compression is the process of transforming information via an encoding that uses fewer bits than the unencoded raw data. Compression is primarily used to reduce the consumption of expensive resources. However there is a trade-off. Compressed data must be decompressed to be viewed. This requires computing resources. So there is trade off between computing resources and file size.

Compression algorithms proceed in two phases. The first involves modelling the original data, the second encoding the symbols using the model. There are many methods for modelling data and for encoding; the two phases are largely independent. Shannon information theory [Shannon, 1948] provided the background theory for encoding, answering the question 'How much can we compress a file without loss?' It turns out that the lower bound for compression is the entropy of the data.

Lossless compression algorithms rely on statistical redundancy, in such a way as to represent the sender's data more concisely, but nevertheless perfectly. Thus lossless data compression will fail if the data doesn't contain a disoverable pattern – random strings are incompressible. Conversely, if the data does contain regularity, then a compression algorithm may make use of the regularity to compress the data. This is the basis of our approach. We apply state-of-the-art lossless compression algorithms, with the aim of detecting regularities – i.e. predictable, repeated structure – in the data (in this case, trees generated by the GP algorithms).

String vs Tree Compression

Both string and tree compression are forms of of lossless compression. String compression can be used to compress any kind of string, such as general English texts. On the other hand, it may not generate good compression when it is used to compress strings representing trees (that is, the prior knowledge that the string represents a tree provides opportunities for compression that string compression algorithms, in general, will find difficult to exploit – we consider tree compression in more detail below). Well known string compression families include dictionary based compression algorithms such as Ziv-Lempel [Ziv & Lempel, 1978] and stochastically-based compression algorithms such as Predict by Partial Match (PPM [Cleary & Witten, 1984].

The Ziv-Lempel method encodes frequently repeated strings by storing a pointer to the first occurrence of the string, thus avoiding encoding every repetition of the string. It is relatively fast – certainly much faster than stochastic compression methods – and its compression efficiency is quite good, though generally not as good as stochastic methods. The Unix gzip utility is a familiar example of a Ziv-Lempel algorithm.

Stochastic methods are more statistically-based than dictionary-based methods. In general, they are able to give higher compression ratios than dictionary-based, but at the cost of computational cost. Since our study is conducted off-line, computational cost is not a major concern, but finding as many regularities as possible in the data is, so we have chosen to use stochastic methods.

PPM is one of the most efficient stochastic algorithms. The coding scheme of PPM uses a Markov model to condition the probability that a particular symbol will occur, conditional 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. For example, suppose an order 2 Markov model is used, and the input sequence so far is 'abracadabr'. Let x be the next symbol (x can be any alphabetic character) Because the order is 2, x is predicted by the context 'br', which has length 2. Of course, 'br' has already appeared in the input sequence, and on that occasion the next character was 'a'. So the PPM model will give high probability to symbol 'a' as the next character. This model has been shown to provide good prediction in many experiments [Cleary & Witten, 1984]. PPM's coding scheme uses adaptive arithmetic coding.

The compression ratio of a stochastic compression method is determined both by the quality of the prediction model, and the encoding algorithm. A good prediction model minimises the error of the prediction of the next symbol. Suppose a model predicts the probability of the next symbol being 0 as 0.99, and the probability of 1 as 0.01. If the input string is in fact 0^{100}, then the information content of the model is $100 * log_2 \frac{100}{99} \approx 1\, bit$. On the other hand, if the input string is 1^{100}, then the information content of the model is $100 * log_2 \frac{100}{1} \approx 460\, bits$.

However our data consists of trees, not strings. String compression algorithms will generally compress trees far from optimally, since they rely on locality *within the string*. However any string representation of a tree must lose some of the locality of the tree. For example, if an inorder representation is used, then leftmost children of a node will be close to the parent in the string representation, but other children, in general, will not. Thus a string compression algorithm is unlikely to find regularities involving children other than the leftmost.

Many tree compression methods aim to replace repeated small subtrees with pointers, analogously to repeated strings in dictionary-based string compression. However finding the smallest tree using replacement is logically equivalent to finding the smallest context-free grammar that generates the string. This problem is known to be NP-Hard [Lehman & Shelat, 2002]. An alternative approach, XMLPPM [Cheney, 2002], conserves the tree structure using a backtracking method.

XMLPPM

XML is, today, probably the best-known format for representing tree structures, and most available tree-compression algorithms use it as the input format. For that reason, we used XML as the representation format for our expression trees. An XML document is composed of element tags which are enclosed by <>. Element tags are divided into start tags $< name >$ and end tags $< /name >$. The start tag can have attributes with values, but this is unnecessary for our application. We need only the element tags to represent the tree. For example, $< a >< b >< c >< /c >< /b >< d >< /d >< /a >$ represents the function (tree) $a(b(c), d)$.

XMLPPM is based on PPM, customised to reflect the tree structure of XML. It uses a backtracking method to archieve tree compression. The backtracking method stores each symbol in the input string to a stack until an end tag appears in the input sequence. When an end tag comes, the algorithm pops the stack once. The stacked symbols are used as context for prediction (the depth of stack used reflects the order of the prediction model). In our example, $< a >$ may be used as context to predict $< d >$ after $< b >< c >< /c >< /b >$ has been encountered.

2 Experiments

2.1 Purpose of Experiments

In our previous work, we hypothesised that the superior performance (when compared with simple tree-based GP) of DEVTAG and DTAG3P on the problems investigated might result from the greater regularity of their genotypes. We based this on a further hypothesis regarding the origin of the repeated code which is manifested in biological genotypes (namely, that this repetition and structure is selected for by developmental evaluation). Our aim in this work is to use compression-based metrics to investigate whether DEVTAG and DTAG3P do, in fact, exhibit greater regularity.

2.2 Methods

Problem Domain

The problem domain is a typical and widely-used class of symbolic regression problems. We can define $F_1 = X$, and then recursively define the class of problems via $F_{i+1} = (F_i * X) + X(i > 0)$. Our use of these problems differs from more general use only in the difficulty of the problem considered (it is unusual to use problems of such high order as the F_9 we investigate here).

Search Space

Four forms of GP (DTAG3P, DEVTAG, TAG3P, GP) are used for comparison. In addition, to separate out the effects of the developmental evaluation selection mechanism and the DTAG3P L-system representation, we introduce a further system, DTAG3PF9ALL (it is not intended as a serious candidate for an evolutionary algorithm, though it may reasonably be seen as a TAG-based analogue of other L-system-based developmental systems). DTAG3PF9ALL is a variant of DTAG3P. It uses the same representation, variation operators etc., differing solely in its evaluation mechanism. Instead of using the incremental evaluation strategy of DTAG3P, it uses standard tournament selection, evaluated on F_9 only (as with TAG3P and standard GP, it is allowed 9 times as many evaluations, to compensate for not receiving evaluation information from $F_1 ... F_9$). Thus DTAG3PF9ALL is directly comparable to GP and TAG3P.

The search space of these algorithms may be defined by a context-free grammar G with binary and unary variable set +, -, *, /, sin and terminal set x.

Formally:
$G = V, T, P, S$
$S = EXP$
$V = EXP, PRE, OP, VAR$
$T = x, sin, +, -, *, /$
P consists of
$EXP \rightarrow EXP\, OP\, EXP | sin\, EXP | VAR$
$OP \rightarrow + | - | * | /$
$VAR \rightarrow x$
Further symbols may appear after simplification (i.e. they are not used in the evolutionary process): 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
$EXP \rightarrow CONST$
$CONST \rightarrow 0|1$
Note that while this grammar is merely used to define the search space in the case of GP, for TAG3P, DEVTAG and DTAG3P the grammar defines the tree representation (that is, the systems actually evolve derivation trees in the corresponding TAG grammar).

Evolutionary Parameters

Parameter settings for the various algorithms are shown in table1 (see [Hoang et al., 2006a] for details).

Simplification

The primary hypothesis underlying this work concerns repeated use of effective code (corresponding to re-use of genes in biological systems). As in biological systems, we anticipated that repetition might also arise in non-effective code, and indeed, that this effect might well dominate the compression metrics. Hence as well as measuring the compression of the raw trees generated by the various evolutionary systems, we also generated the corresponding effective-code subtrees, as detailed in sub-sub-section 1.3, and measured compression on these as well. We used the constraint that a simplification would be accepted if its error, over all 20 sample data points, was less than $\epsilon = 10^{-4}$. Parameter settings for the Tree Simplification are given in table 2.

Table 1. Evolutionary Parameter Settings

Objective	Find a symbolic regression function F_9 (GP, TAG) or $F_1, F_2, \ldots F9$ (DEVTAG, DTAG3P) that fits a given sample of 20 (x_i, y_i) data points
Success Predicate	Sum of errors over 20 points $< \epsilon = 0.01$
Terminal Sets	x – the independent variable
Operators (Function set)	+, -, *, /, sin
Fitness Cases	The sample of 20 points in the interval [-1..+1]
Fitness	Sum of the errors over 20 fitness cases
Genetic Operators	Tournament selection(3); 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
Variation Probabilities	Crossover probability 0.9; mutation probability 0.1
Min/Max initial size for TAG3P, DEVTAG, DTAG3P	2 to 1000
Max depth for GP	20
Number of rules for DTAG3P	4
Min/Max number of β-trees in each successor (DTAG3P)	1 to 5
Population size	250

Table 2. Simplification Parameter Settings

Method	Equivalent Decision Simplification
Relative Error for Simplification	10^{-4}
Subtree search order for Simplification	Breadth-first from leaves
Candidates for substitution	$(0), (1), (x)$

Compression

Each tree under consideration was first converted to XML representation, so that the XMLPPM algorithm could be applied. The compression ratio was calculated as:
(XMLPPM-compressed $size\,of\,tree\,(bits))/((Number\,of\,nodes\,in\,tree) * log_2k)$
(log_2k is the length of bits required to encode each node using arithmetic compression, assuming the terminals are used equally often - i.e. the appropriate prior knowledge - where k is 6 for the terminal set $+, -, *, /, sin, x$ (i.e. the raw expression trees) and 8 for the terminal set $+, -, *, /, sin, x, 0, 1$ (i.e. the simplified expression trees)).

3 Results

Table 3. Success percentages out of 100 runs

GP	TAG	DEVTAG	DTAG3P	DTAG3PF9ALL
0	0	34	81	7

Table 3 shows the success percentage (the proportion of 100 runs) that were successful (i.e. found an exact solution) of each of these algorithms. Apart from DTAG3PF9ALL, these results were reported in [Hoang et al., 2006a]. Clearly, DEVTAG and DTAG3P (the two systems using an incremental/developmental evaluation mechanism) were extraordinarily successful. The design of the developmental evaluation process was based on a hypothesis, previously detailed, about the cause of the systematic and repeated structure of natural genetic systems. Hence it was natural to ascribe the success of the developmental evaluation systems to systematic and repeated structure generated by developmental evaluation.

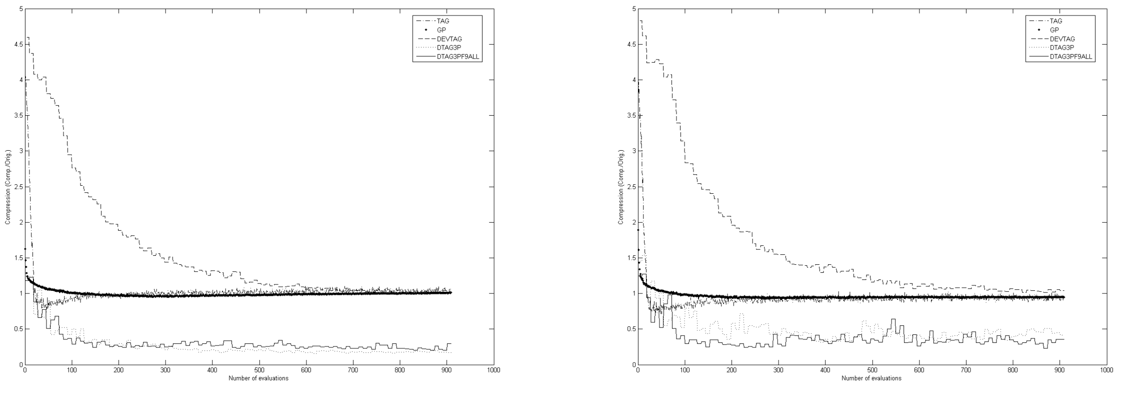

Fig. 1. Compression Ratio of Best Individual vs Evaluations (Left: Original, Right: Effective Code

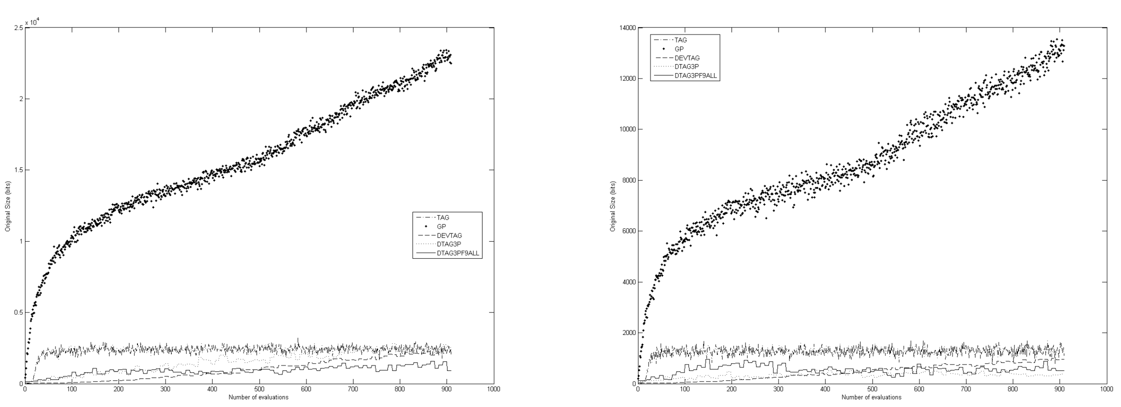

Fig. 2. Size of Best Individual vs Evaluations (Left: Original, Right: Effective Code

Figure 1 shows the compression ratios (i.e. $Compressed\,size/Original\,size$), output by the five algorithms (left hand side) together with the same ratio for the trees as simplified by our Equivalent Decision Simplification algorithm (that is, to a reasonable approximation, the compression ratio for the effective code). In more detail, for each of the 100 runs for each algorithm, we selected the fittest individual for each stage of the algorithm (measured by number of evaluations carried out), and applied simplification and compression processes to that individual. We then calculated the compression ratios for that individual. The points in Figure 1 were obained by averaging these values for a given generation over all 100 runs.

We firstly note that, for three algorithms (GP, TAG, DEVTAG), and for both the original and the simplified trees, the compression ration converges very accurately to 1.0. That is, no compression is achieved for the products of these algorithms.

Secondly, the DTAG3P and DTAG3PF9ALL algorithms converge to compression ratios of around 0.25 (original trees) or just under 0.5 (simplified trees).

Thirdly, in substantial regions of the figures, especially for DEVTAG, the compression ratio is well above 1.0 (that is, there is actually an expansion of the code size on applying XMLPPM). On the hypothesis that this effect might be related to the startup costs of the compression algorithm, we also present, in figure 2, a plot of average size of 'best' individuals vs number of evaluations.

Finally, as a way of understanding the behaviour of the compression algorithm, we used the initialisation stage of the GP algorithm to generate large numbers of random (and hence theoretically incompressible) trees of depths 3, 5 and 7, so that we could see the effect of the compression algorithm on trees where it could, in principle, find only subtree repetitions arising at random.

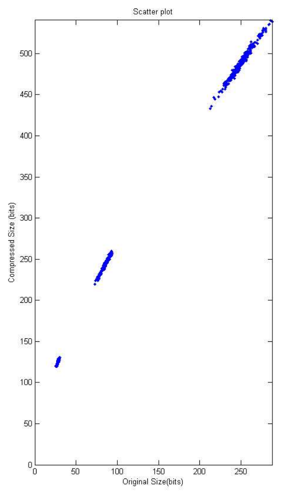

Fig. 3. Compressed vs Original Size for Random Trees

4 Discussion

The convergence of three algorithms (GP, TAG and DEVTAG) to compression ratios of 1.0 was highly surprising to us. It suggests very strongly that there is no repeated code structure available for the XMLPPM compression algorithm to take advantage of (or to be precise, no more than would be expected in a randomly generated tree, so that the cost of representing replications exactly balances the advantage gained by doing so). It was our expectation, before conducting these experiments, that crossover would favour code replication, and that we would therefore see useful compression, especially in the non-effective part of the code (that is, we expected to see repeated introns). These results strongly suggest that repeated introns do not occur more than they would in randomly-generated trees.

Even more surprising, in view of our hypothesis about developmental evaluation, was that there was no hint at all of increased compression arising from it. That is, the two algorithms using developmental evaluation – DEVTAG and DTAG3P – exhibited no more compression (and hence no more replication of code, whether effective or ineffective) than the comparable algorithms. Thus either our hypothesis about developmental evaluation favouring repeated structure is just plain wrong (leaving us to ponder why repeated structure nevertheless does arise in nature), or else the effect is so subtle that the relatively few generations of our runs are not sufficient to display it. Either way, it cannot be the cause of the good relative performance of the developmental evaluation strategies.

On the other hand, the two L-system-based algorithms (i.e. DTAG3P and DTAG3PF9ALL) *do* exhibit increased compression (if anything, slightly more in the latter case), confirming that L-system developmental processes do encourage replicated structure in code.

We suggested that the very large compression ratios (i.e. expansion rather than compression) exhibited by GP and TAG3P right at the start of their runs, and by DEVTAG for most of its run, might be the result of the startup cost of the XMLPPM compression algorithm. This is consistent with the size plots of figure 2, suggesting that most of these effects – and also the decreasing compression ratios exhibited by the DTAG3P and DTAG3PF9ALL algorithms, even though their compression ratios never went above 1.0 – might simply be a result of the difficulty of compressing very small trees. Further confirmation comes from figure 3, showing compression ratios achieved by XMLPPM on various sizes of (incompressible) random trees. We believe it should be possible to extend this analysis, by populating more of figure 3 to estimate the extent of replication more accurately, even at smaller tree sizes, and hence to remove these artefacts of compression algorithms from figure 1.

5 Conclusions and Further Work

The primary conclusion of this paper is that our initial hypothesis, that code replication favoured by developmental evaluation was the primary cause of the good performance of the DEVTAG and DTAG3P algorithms, is plain wrong. We cannot, from these experiments, rule out the importance of developmental evaluation in promoting such structured replication in natural systems – biological evolutionary/developmental systems have had many millions of generations for such an effect to work itself out, compared with the hundreds involved here – but it is quite clear that no such effect has been observed in our experiments. The only thing which promoted code replication in our experiments was the use of an L-system-based developmental method.

This leaves us with two conundrums:

- Why were our developmental evaluation systems (DEVTAG and DTAG3P) so successful in solving this difficult family of problems, if it was not as a result of evolving replicated, structured code?
- Why does nature evolve replicated, structured DNA, and how might we capture this behaviour in artificial systems?

These questions will guide our future work in this area - good answers to either (or both) would be extremely valuable.

On the pragmatic level, we aim to improve the sensitivity of compression-based approach we have used here, by further populating figure 3, and using this information to correct for the bias inherent in the startup costs of compression schemes.

Acknowledgements

This research was financially supported by a Seoul National University support grant for new faculty.

References

[Banzhaf et al., 1998] Banzhaf, Wolfgang, Peter Nordin, Robert E. Keller, & Frank D. Francone 1998. Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications. Morgan Kaufmann, San Francisco, CA, USA.

[Blickle & Thiele, 1994] Blickle, Tobias, & Lothar Thiele 1994. Genetic Programming and Redundancy. In Hopf, J. (ed), Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken), pages 33–38, Im Stadtwald, Building 44, D-66123 Saarbrücken, Germany. Max-Planck-Institut für Informatik (MPI-I-94-241).

[Carroll, 2005] Carroll, S.B. 2005. Endless Forms Most Beautiful: The New Science of Evo Devo and the Making of the Animal Kingdom. W.W. Norton and Company.

[Cheney, 2002] Cheney, J. 2002. Compressing XML with Multiplexed Hierarchical Models. In IEEE Data Compression Conference, pages 163–172, Snowbird, Utah.

[Cleary & Witten, 1984] Cleary, J.G., & I.H. Witten 1984. Data Compression using Adaptive Coding and Partial String Matching. IEEE Transactions on Communications, 32:396–402.

[Cramer, 1985] Cramer, Nichael Lynn 1985. A representation for the Adaptive Generation of Simple Sequential Programs. In Grefenstette, John J. (ed), Proceedings of an International Conference on Genetic Algorithms and the Applications, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA.

[Haddow et al., 2001] Haddow, P.C., Tufte G., & van Remortel P 2001. Shrinking the genotype: L-ystems for Evolvable Hardware. In Evolvable Systems: From Biology to Hardware, 4th International Conference, ICES 2001, Springer Lecture Notes In Computer Science 2210.

[Hoai et al., 2006] Hoai, Nguyen Xuan, R. I. (Bob) McKay, & Daryl Essam 2006. Representation and Structural Difficulty in Genetic Programming. IEEE Transactions on Evolutionary Computation, 10(2):157–166.

[Hoang et al., 2006a] Hoang, T.H., D.L. Essam, R.I. McKay, & X.H. Nguyen 2006a. Developmental Evaluation in Genetic Programming: A TAG-Based Framework. In Procceedings of the third Asia-Pacific Workshop on Genetic Programming, VietNam Military Technical Academy, Hanoi, VietNam.

[Hoang et al., 2006b] Hoang, Tuan-Hao, Daryl Essam, R. I. (Bob) McKay, & Xuan Hoai Nguyen 2006b. Solving Symbolic Regression Problems using Incremental Evaluation in Genetic Programming. In Proceedings of the 2006 IEEE Congress on Evolutionary Computation, pages 7487–7494, Vancouver. IEEE Press.

[Hornby, 2003] Hornby, Gregory S. 2003. Generative Representations for Evolving Families of Designs. In Cantú-Paz, E., J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, & J. Miller (eds), Genetic and Evolutionary Computation – GECCO-2003, volume 2724 of *LNCS*, pages 1678–1689, Chicago. Springer-Verlag.

[Jacob, 1994] Jacob, Christian 1994. Genetic L-System Programming. In Davidor, Yuval, Hans-Paul Schwefel, & Reinhard Männer (eds), Parallel Problem Solving from Nature III, volume 866 of *LNCS*, pages 334–343, Jerusalem. Springer-Verlag.

[Joshi et al., 1975] Joshi, A.K., L.S. Levy, & M. Takahashi 1975. Tree Adjunct Grammars. J. Comput. Syst. Sci., 10:136–163.

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

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

[Lehman & Shelat, 2002] Lehman, E., & A. Shelat 2002. Approximations Algorithms for Grammar-Based Compression. In Thirteenth Annual Symposium on Discrete Algorithms (SODA'02).

[Lindenmayer, 1968] Lindenmayer, A. 1968. Mathematical models for cellular interaction in development, parts I and II. Journal of Theoretical Biology, 18:280–299 and 300–315.

[McKay et al., 2006] McKay, Robert Ian, Tuan Hao Hoang, Daryl Leslie Essam, & Xuan Hoai Nguyen 2006. Developmental Evaluation in Genetic Programming: the Preliminary Results. In Collet, Pierre, Marco Tomassini, Marc Ebner, Steven Gustafson, & Anikó Ekárt (eds), Proceedings of the 9th European Conference on Genetic Programming, volume 3905 of *Lecture Notes in Computer Science*, pages 280–289, Budapest, Hungary. Springer.

[Mori et al., 2007] Mori, N., R.I. McKay, X.H. Nguyen, & D.L. Essam 2007. How Different are Genetic Programs: New Methods for Studying Diversity and Complexity in Genetic Programming. in preparation.

[Schlosser & Wagner, 2004] Schlosser, G., & G.P. (eds) Wagner 2004. Modularity in Development and Evolution. University of Chicago Press, Chicago, Ill, USA.

[Shannon, 1948] Shannon, C.E. 1948. A Mathematical Theory of Communication. The Bell System Technical Journal, 27:379–423 and 623–656.

[Ziv & Lempel, 1978] Ziv, J., & A. Lempel 1978. Compression of Individual Sequences Via Variable-Rate Coding. IEEE Transactions on Information Theory, 24:530–536.