Analysing the Regularity of Genomes using Compression and Expression Simplification

Jungseok Shin[1], Moonyoung Kang[1], R I (Bob) McKay[1], Xuan Nguyen[3], Tuan-Hao Hoang[2], Naoki Mori[4], and Daryl Essam[2]

[1] Structural Complexity Laboratory, Seoul National University, Seoul 151744 Korea
[2] Australian Defence Force Academy, Campbell ACT 2601
[3] Natural Computation Group, VietNam Military Technical Academy, Hanoi
[4] Osaka Prefecture University, Osaka, Japan
shin119@yahoo.com,yerihyo@hotmail.com,hao_hth@yahoo.com,rim@cse.snu.ac.kr
mori@cs.osakafu-u.ac.jp,nxhoai@gmail.com,d.essam@adfa.edu.au}
http://sc.snu.ac.kr

Abstract. We propose expression simplification and tree compression as aids in understanding the evolution of regular structure in Genetic Programming individuals. We apply the analysis to two previously-published algorithms, which aimed to promote regular and repeated structure. One relies on subtree duplication operators, the other uses repeated evaluation during a developmental process. Both successfully generated solutions to difficult problems, their success being ascribed to promotion of regular structure. Our analysis modifies this ascription: the evolution of regular structure is more complex than anticipated, and the success of the techniques may have arisen from a combination of promotion of regularity, and other, so far unidentified, effects.

Key words: Genetic Programming, compression, simplification, ppm

1 Introduction

Modularity has been heavily studied in Genetic Programming (GP), emphasising function-call or -expansion models based in Computer Science [1–3], such as Koza's Automatically Defined Functions. In biological DNA systems, modularity such as in the homeobox complex arises primarily through repetition and variation of genes [5]. This difference is reflected in terminology, where 'modularity' in Computer Science implies encapsulation and re-use, but has no such connotation in Biological Sciences. In this paper, we emphasise the biological form – repetition and variation of sub-structures (in GP terms, re-use of building blocks). To reduce confusion, we eschew the term 'modularity' from this point.

Our main idea is to use compression algorithms to estimate the regularity of genotypes. To the best of our knowledge, this has not previously been done, though there is some hint in Lathrop's work [6] relating compression depth to GP behaviour, and in Svangard and Nordin's [7] use of compression to estimate similarity between, rather than self-similarity within, genotypes. Compression methods (sometimes implicitly) build a model of the structural regularities in data, then use the model to build a compressed representation of the data. Hence any regularities that are represented in the model lead to increased compression. If we measure the compression, we are implicitly measuring the extent of regularity (or at least, the extent to which that regularity matched the model).

It is well-known that GP generates redundant code [8] – expression trees which may be replaced by smaller trees with the same evaluation. In this study, we are interested in regularities both in the whole code, and in the effective part of the code. Regularities may arise in the ineffective code through a variety of mechanisms, yet have no useful effect on learning. Analogously, DNA sequences contain huge segments of repeated pairs in non-coding regions, which have no known effect on the resulting biology. An analysis focusing only on overall measures of regularity – inevitably statistically dominated by the simple repetitions – would miss the far more important regularites occurring through repetition of large segments of coding regions (genes) and equally important, their regularly structured variation. Hence we are interested in analysing the compression, not only of trees, but of their effective backbones.

We would like to simplify expression trees to the simplest equivalent form. For many function domains of interest, this is a known uncomputable problem. The best we can hope is to reduce the subtrees to a near-minimal form. For this purpose, we use the Equivalent Decision Simplification (EDS [9]) method, specialised to the arithmetic domain of our experiments.

The remainder of this section gives a brief introduction to the necessary background in compression and expression simplification. Section 2 introduces our compression-based metric for comparing the regularity of trees, and two previously-studied algorithms in which improvements in performance have been ascribed to re-use of building blocks. We report on the results of our new analyses in section 3, discussing the ramifications in section 4. In section 5, we present our conclusions about the effectiveness of compression-based methods to gain understanding of the behaviour of GP systems.

1.1 Compression

Data compression is the process of encoding information in a form that uses fewer bits than the raw data. Compression algorithms proceed in two phases, modelling the original data, and encoding the symbols using the model.

Lossless compression algorithms rely on statistical redundancy, using it to represent the sender's data more concisely, but nevertheless perfectly. Thus lossless data compression will fail if the data doesn't contain a discoverable 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 tree compression algorithms, with the aim of detecting regularities – i.e. predictable, repeated structure – in the data (in this case, trees generated by GP algorithms).

String vs Tree Compression Stochastically-based compression algorithms such as Predict by Partial Match (PPM [10] generally provide higher compression ratios than the better-known dictionary-based methods such as Ziv-Lempel [11], but at the cost of speed. Since this study was conducted off-line, speed was not a primary issue, so stochastic methods were used. XMLPPM [12] extends the PPM model to compress trees rather than strings. It does so using a stack to record the context at all branch points. XMLPPM uses XML format to represent the input trees, rather than more economical tree representations; but this is merely a technical detail. In our experiments, we record the original tree size in a direct (inorder) representation, then convert trees to XML format for compression.

1.2 Simplification

Rule-based Simplification Previous simplification work [13–15] used syntactic methods to simplify trees; but finding all syntactic redundancies is undecidable. This point is not merely theoretical. If the subtreee T is complex, syntactic algorithms may find $(T-1)+(1-T)$ difficult to simplify.

Equivalent Decision Simplification A stronger method, Equivalent Decision Simplification (EDS), has recently been proposed [9]. In EDS, 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. For this problem domain, EDS found far more simplifications than rule-based simplification. The results strongly suggested that EDS was finding essentially all the simplifications available (to be precise, the behaviour of simplified-genotype entropy, and of phenotype entropy, were virtually identical).

The problem remains, of finding an appropriate tree for substitution. We solve this by making single terminal nodes the candidates for substitution. By recursively simplifying each subtree, the whole tree is simplified. EDS is used to simplify trees in this experiment, so that we can investigate the behaviour of the effective code.

2 Methods

Using PPM-based tree compression and EDS, we re-analyse data from two previous sets of experiments which involved hypotheses about the replication of building blocks. Both were based on the TAG representation [16]. The details of the representation are unimportant here; its key properties are

- TAG representation is based on labelled trees, as in Koza-style GP
- the system uses subtree crossover and mutation, as in Koza-style GP
- Any rooted subtree of a valid TAG tree, or any consistent extension, is valid

Because of the last property (which we call 'feasibility'), it is possible to extend TAG3P in many ways. In this paper, we consider two such extensions:

– Extending TAG3P with subtree duplication and truncation operators
– Extending TAG3P with a developmental evaluation process

2.1 Compression and Small Trees

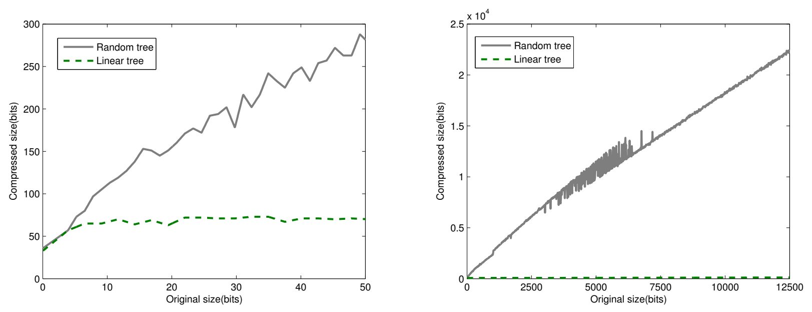

Fig. 1. Compression of Random and Linear Trees (left: small trees; right: large)

While compression algorithms usually compress large objects well, they generally have a startup cost, with difficulty in compressing small objects. Thus relative compression of two subtrees of the same size gives useful information about their relative regularity, but can be misleading for trees of very different sizes. Instead, we report the relative compression, relative to two extremes. At one extreme, random trees are incompressible; on the other, XMLPPM is especially effective at compressing linear trees. We report the relative compressed size of an individual, relative to random and linear trees. In detail, if a tree of size S compresses to size C, we first compute the maximum compressed size R of random trees, and L of linear trees, originally of size S, and we report the ratio $(C - L)/(R - L)$. The metric has the following desirable properties:

– For trees of a given original size, it is monotonic with the compression (and by extension, estimates the degree of replication)
– It discounts any size-dependent start-up cost of the compression algorithm

The random trees were generated by half-and-half initialisation. Figure 1 shows the compressed size (Y axis) vs original size (X axis). Note the high degree of nonlinearity at small sizes, and the step in compression of random trees at an original size of between 5,000 and 10,000. The latter is probably an artefact of the fixed-size context used by PPM algorithms. An alternative algorithm, PPM*, uses unbounded context, but has never been implemented for tree-compression. We hope to report on its use in the near future.

While the genotype representations considered here differ widely, 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.

2.2 Duplication/Truncation Operators

In the duplication operator, a random node in the current tree is chosen. The subtree rooted at that node is copied. A random leaf node in the tree is then chosen. The copied subtree is attached at that leaf node.

The duplication operator expands the size of the TAG3P tree. Used alone, it causes bloat so rapid as to prevent useful evolution. Its use is balanced by another operator, truncation. Truncation also randomly selects a location in the current tree. Instead of copying the subtree rooted at that node, truncation removes it. Duplication and truncation operators were applied at the same rate, to balance their effects on size. While their interaction with selection is difficult to analyse, in practice bloating was seen to be roughly similar to that observed in the basic TAG3P system, and typical of a tree-based GP system. Duplication was expected to be a useful operator for problems with highly regular forms, such as the well-known polynomial regression problem, in which GP is used 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. The grammar in table 1 defines the solution space. 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: $EXP \rightarrow CONST$ and $CONST \rightarrow 0|1$.

Table 1. Grammar describing Solution Space

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

Our expectations were borne out in experiments reported in [16]. Duplication improved performance when used as a mutation operator in TAG3P, and more substantially when used as a local search operator. However the underlying assumption, that use of duplication had promoted duplicated code segments appropriate for the domain, was not tested. We compared five algorithms (TAG: the basic TAG3P system; TAGCROSS: TAG3P with no mutation operator; TAGM: TAG3P with duplication/truncation as mutation operators; LSTAG10 and LSTAG50: TAG3P with duplication/truncation as local search operators – 10 and 50 steps respectively). All algorithms had a fixed budget of evaluations, so more local search corresponded to fewer evolutionary generations.

2.3 Developmental Evaluation

Developmental evaluation [17] has been proposed as a mechanism to promote structural regularity in GP. It uses a typical developmental process, but relying on the feasibility property of TAG3P, evaluates and selects individuals during development, in analogy with biological systems. In more detail, individuals are evaluated on a family of problems of increasing difficulty as they develop, with performance at earlier stages favoured over performance at later stages. In experiments using the family of functions $F_1, ...F_9$, and the grammar from table 1, developmental systems achieved excellent performance (two variants were used, DEVTAG using a trivial incremental growth process, and DTAG3P, an L-systems developmental model; in the experiments, they were compared with standard tree-based GP (DE), TAG3P, and DTAG3PF9ALL: using DTAG3P L-system representation, but standard evolutionary evaluation). Recently [18], we reported on comparisons between these systems using direct compression measurements, and noted the difficulty of comparing compression ratios for trees of differing sizes. We overcome this with the ratio method from section 2.1.

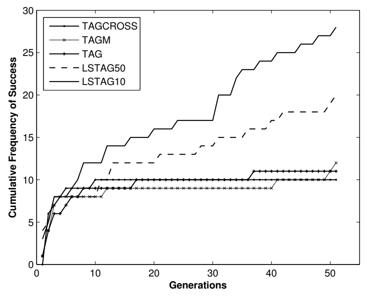

Fig. 2. Cumulative Success Ratio vs Evaluations ($*10^4$)

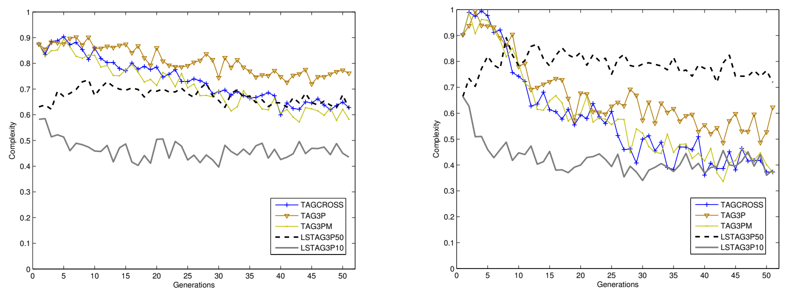

Fig. 3. Duplication: Compression Complexity vs Evaluations ($*10^4$) (left: original; right: effective code)

3 Results

3.1 Duplication/Truncation

Figure 2 shows the success ratio (out of 100 runs) for the different algorithms. As previously reported, using crossover only, or duplication/truncation as mutation operators, makes little difference to the overall performance of the underlying TAG3P algorithm (which performs slightly better than standard GP). Local search for 10 steps of duplication and truncation substantially improves performance, but too much local search degrades it.

Figure 3 shows the complexity of the fittest individual in each generation, using our measure. We note that individual complexity tends to decrease throughout a run, both for the original code and the effective backbone; and that 10 steps of local search give the lowest complexity for either original or effective code, but 50 steps give the highest for the latter.

3.2 Developmental Evaluation

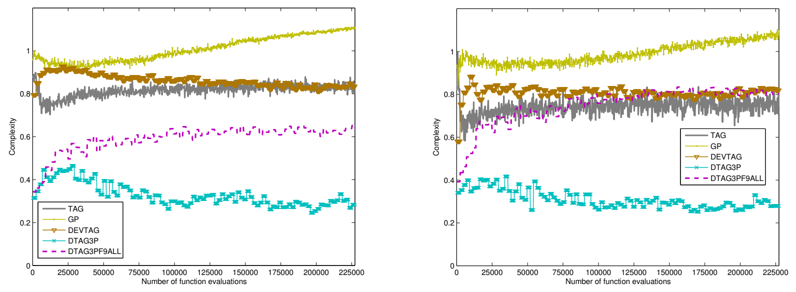

Fig. 4. Developmental Evaluation: Compression Complexity vs Generations (left: original; right: effective code)

Previously presented results have shown the very high success rates of the DTAG3P and DEVTAG algorithms relative to the underlying TAG algorithm on this problem (around 70% and 30% success respectively, for the same number of potential evaluations as used in subsection 3.1). For the same budget of evaluations, standard tree-based GP has essentially zero probability of success.

Figure 4 shows the compression-complexity of the algorithms plotted against generations. We note the very low replication in both overall and effective code components of standard GP; and the high high replication in both overall and effective code components of DTAG3P. As far as code complexity is concerned, the DEVTAG algorithm performs almost identically to the underlying TAG3P algorithm. Finally, L-system development on its own (DTAG3PF9ALL) promotes code replication at the overall code level, but without developmental evaluation, this replication is not reflected in the effective code.

4 Discussion

4.1 Duplication/Truncation

The results are partially consistent with our original hypothesis, that code replication would help to solve the F_9 problem. 10 steps of local search with duplication and truncation improve the probability of success, and increase the amount of code repetition, both in original and effective code. But this is not the whole story. 50 steps of local search degrade the effectiveness of the algorithm, but not so far as to be worse than the non-local-search versions (perhaps 50 steps of local search do not allow enough generations of evolutionary search for effective exploration). Yet 50 steps of local search give the lowest level of effective code replication. Thus success is not simply correlated with replicated effective code. This leads on to a range of potential hypotheses. Perhaps truncation is more important to the success of the success of the duplication/truncation local search combination than we had supposed. Perhaps what is replicated is more important than the degree of replication. Compression analysis has opened up a range of new questions about this matter.

4.2 Developmental Evaluation

Similarly, the Developmental Evaluation results strongly support the underlying hypothesis, that a developmental process, and evaluation during development, are both necessary for promoting repetitively structured effective code, such as arises in DNA. Interestingly, this contrasts strongly with our earlier results (using simple compression ratios, rather than the more size-independent method used here), which seemed to disprove this hypothesis. L-system-based development alone, without developmental evaluation (DTAG3PF9ALL), initially generates a high degree of regularity, but this regularity is rapidly lost from the effective code, and more gradually from the overall code.

However this is not the whole story. DEVTAG (which uses a trivial developmental process, but still uses incremental evaluation during development) performs well on this problem – far better than DTAG3PF9ALL – yet incremental evaluation does not promote regularity at all in this case (the individuals are, if anything, less regular than those generated by the basic TAG algorithm).

5 Conclusions

In sections 4.1 and 4.2, previously accepted hypotheses about the mode of action of particular evolutionary systems were tested using the twin tools of EDS and compression. In both cases, in their broadest outlines, the hypotheses were borne out by these analyses. In more detail, they raised intriguing new questions. In both cases, the success of the proposed modifications does seem to reflect increased regularity of the effective code; however this is not the whole story, and the effectiveness of the systems is due, in part, to causes other than simple replication of good building blocks. This has opened up new lines of inquiry, investigating what has allowed these systems to perform so effectively.

We don't believe this situation is unique to these two specific problems. These techniques can help to understanding the behaviour of evolutionary algorithms in any domain where replication of building blocks may be important (and in particular, to understanding the level of success achieved by modularising approaches such as ADFs). But we don't believe the applications stop there. In these experiments, we compressed individual trees, to determine the level of replication within trees. It is equally straightforward to compress whole populations, and use this compression to estimate the level of replication of code segments across the population. Such analyses should allow us to gain a deeper understanding of the role of building blocks, since any promotion of building blocks should result in regularities in effective code that compression algorithms should be able to discover and exploit.

The compression techniques are not problem- or representation- specific, so they are readily extended to new systems and problem domains. The principle of EDS is equally extensible, at least to problem domains where a reasonable definition of 'equivalent decision' is available – for example to other arithmetic function sets. However in some domains, it may be unnecessary. in Boolean domains, EDS reduces to model-theoretic (i.e. truth- table) methods, and syntactic simplification to proof-theoretic methods. For perfect accuracy, the former are exponential time, while the latter are NP-complete. It is not yet clear which will yield better performance if some level of missed equivalences is acceptable.

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

References

1. Koza, J.R.: Hierarchical automatic function definition in genetic programming. In Whitley, L.D., ed.: Foundations of Genetic Algorithms 2, Vail, Colorado, USA, Morgan Kaufmann (24–29 July 1992) 297–318
2. Spector, L.: Evolving control structures with automatically defined macros. In Siegel, E.V., Koza, J.R., eds.: Working Notes for the AAAI Symposium on Genetic Programming, MIT, Cambridge, MA, USA, AAAI (10–12 November 1995) 99–105
3. Woodward, J.R.: Modularity in genetic programming. In Ryan, C., Soule, T., Keijzer, M., Tsang, E., Poli, R., Costa, E., eds.: Genetic Programming, Proceedings of EuroGP'2003. Volume 2610 of LNCS., Essex, Springer-Verlag (14-16 April 2003) 254–263
4. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge Massachusetts (May 1994)
5. Schlosser, G., Wagner, G.e.: Modularity in Development and Evolution. University of Chicago Press, Chicago, Ill, USA (2004)
6. Lathrop, J.I.: Compression depth and genetic programs. In Koza, J.R., Deb, K., Dorigo, M., Fogel, D.B., Garzon, M., Iba, H., Riolo, R.L., eds.: Genetic Programming 1997: Proceedings of the Second Annual Conference, Stanford University, CA, USA, Morgan Kaufmann (13-16 July 1997) 370–379

7. Svangard, N., Nordin, P.: Automated aesthetic selection of evolutionary art by distance based classification of genomes and phenomes using the universal similarity metric. In Raidl, G.R., Cagnoni, S., Branke, J., Corne, D.W., Drechsler, R., Jin, Y., Johnson, C.R., Machado, P., Marchiori, E., Rothlauf, F., Smith, G.D., Squillero, G., eds.: Applications of Evolutionary Computing, EvoWorkshops2004: EvoBIO, EvoCOMNET, EvoHOT, EvoIASP, EvoMUSART, EvoSTOC. Volume 3005 of LNCS., Coimbra, Portugal, Springer Verlag (5-7 April 2004) 447–456
8. Blickle, T., Thiele, L.: Genetic programming and redundancy. In Hopf, J., ed.: Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken), Im Stadtwald, Building 44, D-66123 Saarbrücken, Germany, Max-Planck-Institut für Informatik (MPI-I-94-241) (1994) 33–38
9. Mori, N., McKay, R., Nguyen, X., Essam, D.: How different are genetic programs: New methods for studying diversity and complexity in genetic programming. in preparation (2007)
10. Cleary, J., Witten, I.: Data compression using adaptive coding and partial string matching. IEEE Transactions on Communications **32** (1984) 396–402
11. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory **24** (1978) 530–536
12. Cheney, J.: Compressing xml with multiplexed hierarchical models. In: IEEE Data Compression Conference, Snowbird, Utah (2002) 163–172
13. Hooper, D., Flann, N.S.: Improving the accuracy and robustness of genetic programming through expression simplification. In Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L., eds.: Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University, CA, USA, MIT Press (28–31 July 1996) 428
14. Ekart, A.: Shorter fitness preserving genetic programs. In Fonlupt, C., Hao, J.K., Lutton, E., Ronald, E., Schoenauer, M., eds.: Artificial Evolution. 4th European Conference, AE'99, Selected Papers. Volume 1829 of LNCS., Dunkerque, France (3-5 November 2000) 73–83
15. Wong, P., Zhang, M.: Algebraic simplification of genetic programs during evolution. Technical Report CS-TR-06-7, Computer Science, Victoria University of Wellington, New Zealand (2006)
16. Hoai, N.X., McKay, R.I., Essam, D., Hao, H.T.: Genetic transposition in tree-adjoining grammar guided genetic programming: The duplication operator. In Keijzer, M., Tettamanzi, A., Collet, P., van Hemert, J.I., Tomassini, M., eds.: Proceedings of the 8th European Conference on Genetic Programming. Volume 3447 of Lecture Notes in Computer Science., Lausanne, Switzerland, Springer (30 March - 1 April 2005) 108–119
17. Hoang, T., Essam, D., McKay, R., Nguyen, X.: 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 (October 12-14 2006)
18. Kang, M., Shin, J., Hoang, T., McKay, R., Essam, D., Mori, N., Nguyen, X.: Code duplication and developmental evaluation in genetic programming. In: 10^th Asia-Pacific Workshop on Intelligent and Evolutionary Systems, Seoul, Korea (2006)