How functional dependency adapts to salience hierarchy in the GAuGE system

Miguel Nicolau and Conor Ryan

Department Of Computer Science And Information Systems
University of Limerick
Ireland
{Conor.Ryan|Miguel.Nicolau}@ul.ie

Abstract. GAuGE is a position independent genetic algorithm that suffers from neither under nor over-specification, and uses a genotype to phenotype mapping process. By specifying both the position and the value of each gene, it has the potential to group important data together in the genotype string, to prevent it from being broken up and disrupted during the evolution process. To test this ability, GAuGE was applied to a set of problems with exponentially scaled salience. The results obtained demonstrate that GAuGE is indeed moving the more salient genes to the start of the genotype strings, creating robust individuals that are built in a progressive fashion from the left to the right side of the genotype.

1 Introduction

GAuGE [13], *Genetic Algorithms using Grammatical Evolution*, is a recently introduced position independent Genetic Algorithm, built using many of the principles of Grammatical Evolution [12]. GAuGE achieves position independence by encoding each gene as a pair of values, the first coding the position or *locus* of the gene, and the second its value or *allele*.

One of the interesting properties of GAuGE appears to be its ability to prioritise information finding. That is, given its position independence, it should be able to move more important genes to the start of a chromosome. This means that even if genes in the original GA representation are geographically disparate, functionally linked genes can be grouped together at the start of the chromosome.

This paper uses a set of standard benchmark problems to test this hypothesis. We take the well known BinInt [11] problem, and a variation which we call InvBinInt, and demonstrate that the system consistently places the most salient genes at the start of the chromosome, where they are less likely to be broken up.

This paper consists of seven sections. Section 2 contains an overview of the Grammatical Evolution system. Section 3 presents the GAuGE system, and includes an example of the mapping process employed, as well as a brief report of previous results obtained with the system. Section 4 presents the problems used in the experiments. Section 5 describes the setup used with those experiments, and analyses the results obtained. Finally, Section 6 concludes the work, and Section 7 outlines some possible future lines of work.

2 Grammatical Evolution

Grammatical Evolution (GE) is an evolutionary automatic programming type system, which evolves strings of binary values and uses a BNF (Backus-Naur Form) grammar to map those strings into programs. This mapping process involves transforming the binary individual into a string of integer values, and using those values to choose transformations from the given grammar. A given start symbol will then be mapped onto a syntactically correct program, by applying the chosen transformations.

The mapping process employed in GE is based on the idea of a genotype to phenotype mapping: an individual comprised of binary values (genotype) is evolved, which, before being evaluated for fitness, has to undergo a mapping process that will create a functional program (phenotype), which is then evaluated by the given fitness function. This process separates two distinct spaces, a search space and a solution space.

Another feature in GE is the use of degenerate genetic code [10]: by using the *mod* function to normalise each integer to a finite number of production rules from the grammar, different integer values can select the same production rule. This creates a many-to-one mapping from a genotype string to a phenotype individual, which means that the genotype can be modified without necessarily changing the phenotype, through a process known as neutral mutation [5, 1].

Finally, the production rule chosen by each of the values in the integer string is dependent on the values preceding it, as those values determine which non-terminal symbols remain to be mapped in the current individual. This creates a functional dependency between each gene and those preceding it, which in turn guides each individual to be built from the leftmost genes to the rightmost ones, and helps the individual in preserving a good structure in its left-hand side during the evolution process, when it is submitted to the harsh effects of operators like crossover. This has been termed the "ripple effect" [4].

3 Genetic Algorithms using Grammatical Evolution

GAuGE (Genetic Algorithms using Grammatical Evolution) [13] is based on many of the same ideas as GE. It uses a genotype to phenotype mapping in much the same fashion: an individual is composed of a binary sequence (genotype) that, once ready for evaluation, is mapped onto a string of integer values; these are however interpreted as a sequence of *(locus, allele)* pairs, which are then used to finally build a new binary string (the phenotype), ready to be evaluated.

Another feature of GE present in GAuGE is the functional dependency of genes within an individual, i.e. the function of each gene is dependent on those preceding it. Recent work has seen this dependency extended, and a new system, LinkGAuGE [7], was used to successfully solve a set of hard deceptive problems.

Since the position and value of each bit of the phenotype string are expressed on each gene, geographically disparate values of the phenotype can be grouped together on the genotype. This leads to the creation of tight building blocks at the start of the genome that can be gradually grown by the evolutionary process.

Work by Bean [2] with the Random Keys Genetic Algorithm (RKGA) hinted that a tight linkage between genes would result in both a smoother transition between parents and offspring when genetic operators are applied, and an error-free mapping to a sequence of ordinal numbers. More recently, Harik [3] has applied the principles of functional dependency in the Linkage Learning Genetic Algorithm (LLGA), in which a chromosome is expressed as a circular list of genes, with the functionality of a gene being dependent on a chosen interpretation point, and the genes between that point and the current gene.

3.1 Example GAuGE Mapping

To illustrate the mapping process employed in GAuGE, we will use as an example individual the following binary sequence:

```
0111 0001 0001 0100 0001 1001 0010 0011
```

The first step is to map it onto an integer string. For the purpose of brevity, we will use four bits to encode each integer (rather than the eight used in the actual experiments) and therefore end up with the following string:

```
7 1 1 4 1 9 2 3
```

This string will be evaluated as a sequence of four *(locus, allele)* pairs, and will be used to fill in a string of four bits. We start by taking the first position specified, `7`, and map it onto the number of available positions in the final string (i.e., `4`), by calculating the remainder of the division of `7` by `4` ($7\%4$), giving the value `3` (i.e., the fourth position in the phenotype string). We use a similar process to map the value for that position, `1`, into a binary value: $1\%2 = 1$. This is the state of the final array after the above steps are executed:

```
? ? ? 1
```

By taking the next pair, `(1,4)`, we again map the position onto the number of available positions, in this case `3`, which gives us $1\%3 = 1$ (second free position), and normalize the value `4` onto a binary value, which gives us $4\%2 = 0$:

```
? 0 ? 1
```

With the next pair, `(1,9)`, we map the position `1` onto the number of available positions, `2`, by calculating $7\%2 = 1$ (second free position, which is the third position in the string), and the value `9` onto a binary value, $9\%2 = 1$:

```
? 0 1 1
```

Finally, with the last pair, we map the position `2` onto the number of remaining places, in this case `1`, giving the value $2\%1 = 0$, and place the value $3\%2 = 1$ in it (note that the last position will always be mapped onto value `0`, since there is only one free position left in the final individual. Our phenotype, now ready for evaluation, is the string:

```
1 0 1 1
```

3.2 Previous results

In previous experiments [13], GAuGE was applied to both a standard genetic algorithm problem (Onemax) and a new deceptive ordering problem. On the former, its performance was as good as that of a simple genetic algorithm, showing that its overhead processing (namely its mapping process) does not result in a loss of performance in simple problems, while on the latter, its *(locus,allele)* specification was shown to provide the flexibility of swapping elements in a solution, helping the system to avoid local optima. More recently, the functional dependency seen in the position specification was extended to the value specifications, and the resulting system was shown to have the potential to solve hard deceptive problems [7].

4 Problems

In our experiments, our aim was not to measure the performance of the system on a set of problems, but rather to understand its dynamics.[1] More specifically, the hypothesis we aimed to test is that, because of its distinct *locus* and *allele* specification, more important genes tend to be moved to the start of the genotype, whereas less important genes (or more volatile ones) tend to be pushed to the end of the genotype. In other words, the salience structure of a problem can establish a hierarchy of relevance for the genes of each individual, and GAuGE automatically models that hierarchy on its genotype string.

The following problems were therefore specifically chosen to test our hypothesis, and rather than concentrating on the success or otherwise of GAuGE on those problems, the emphasis is on the way the system handles (and possibly exploits) salience at no extra computational cost.

4.1 BinInt

The BinInt problem is an exponentially scaled problem, defined by the formula:

$$f(x) = \sum_{i=0}^{l-1} x_i 2^{l-i-1} \qquad x_i \in \{1,0\}$$

where l is the string length, and x_i the allele at position i (with positions in a phenotype string ranging from 0 to $l-1$). This problem has the interesting characteristic that the salience of each allele is higher than the combined marginal fitness contributions of all the following alleles. This means that within an individual (phenotype), the fitness contributions decrease from left to right. Table 1 shows some examples of fitness evaluations.

This problem was first introduced by Rudnick [11], where it was used to investigate the phenomenon of *domino convergence* (the convergence speed of building blocks). Its convergence time complexity was later measured by Thierens et al. [15], to illustrate the idea of the temporal-salience structure of problems.

[1] The system does solve the problems presented, but that is beyond the scope of this paper.

Table 1. Example fitness evaluations for a series of binary strings of length 8 using the BinInt problem, illustrating the salience of leftmost allele.

Binary string	Fitness value
01011111	95
01111101	125
01111111	127
10000000	128
11111111	255

4.2 InvBinInt

As a second problem we introduce the InvBinInt problem, a variation of the BinInt problem in which an individual is evaluated as an inverted binary number, i.e. encoded from right to left. It is defined by the formula:

$$f(x) = \sum_{i=0}^{l-1} x_i 2^i \qquad x_i \in \{1, 0\}$$

4.3 Onemax

This is a well known genetic algorithm problem, defined by the formula:

$$f(x) = \sum_{i=0}^{l-1} x_i \qquad x_i \in \{1, 0\}$$

In this problem, all *alleles* are equally salient, and the convergence time is uniform for all genes [15]. This means that the mutation of an *allele* will always have the same fitness impact, regardless of its *locus*.

This problem was deliberately chosen because all *alleles* are equally salient; with a uniform distribution of salience, GAuGE should not establish a hierarchy on its *locus* specifications.

5 Experiments

5.1 Experimental Setup

For all three problems, we used the same standard setup that has been used with GAuGE on previous occasions [13, 7]. This is a steady-state replacement strategy with roulette-wheel selection, probability of one-point crossover (called *ripple crossover* [4], because of the mentioned ripple effect) of 0.9, and probability of (point) mutation of 0.01. Each of the position and value fields were encoded using 8 bits. Several population sizes were used, of 50, 100, 200, 400, 800 and 1600 individuals, and all experiments were ran 100 times, with different random seeds. Finally, phenotype string lengths of 8, 16, 24, 32, 40, 48, 56 and 64 were used with each problem.

5.2 Results

The results for all experiments are plotted in the graphs shown in Figures 1, 2 and 3. These graphs were plotted by taking the best individual of each run, and averaging their *locus* specifications; the x-axis shows each gene in the genotype string, whereas the y-axis shows the *locus* specified in each gene, averaged over the 100 runs. For each graph, a least-squares regression equation for the data points was used to draw a line, with its label being the slope of that line.

BinInt The results for this experiment show that GAuGE identifies the leftmost values in the phenotype to be the most salient ones, and therefore the best individuals tend on average to encode the first positions of the phenotype within the first genes of the genotype.

InvBinInt In this problem, the opposite effect was observed, i.e. because the rightmost values of the phenotype string are the more salient ones, these tend to be encoded by the leftmost genes in the genotype string. GAuGE seems therefore to effectively build its genotype strings from the left to the right-hand side.

Onemax In this final set of experiments, all *loci* in the phenotype string have the same salience. The results obtained show that GAuGE does not elect any particular *locus* to be encoded in the first genes of the genotype string, if there is no difference of salience for each of the values of the phenotype.

5.3 Analysis

From the results obtained, it can be seen that GAuGE is effectively encoding the most salient values of the phenotype string in the left side of its genotype strings, and then building those strings from left to right. No mechanisms were employed to prefer these individuals, other than a standard roulette wheel selection.

This fits nicely with the *ripple crossover* as used in GAuGE, and with the functional dependency used in the specification of *loci* in each gene. By preferring individuals that encode the more salient values on their leftmost genes, genotype strings are effectively built from left to right. As the best individuals in the population tend to encode the more salient genes on their left side, and because they will be combined with each other, the ripple effect will exchange good building blocks, as their left-hand side context will be similar (or identical).

Although the more salient *loci* tend to be stored in the left side of the genotype, GAuGE does not require this ordering at the genotype level to solve the problem, as implied by the results: in the BinInt experiments, for example, the first gene is not always coding for position 0 of the phenotype; rather there is a tendency to move the more salient genes to the start of the genotype string. This suggests in turn that there is enough diversity at the genotypic level, and that GAuGE does not have to order its genes to solve problems of varying salience.

Fig. 1. Results for the BinInt problem. The graphs plot the average *locus* specification of each gene in the genotype string, averaged over 100 runs, for different problem lengths.

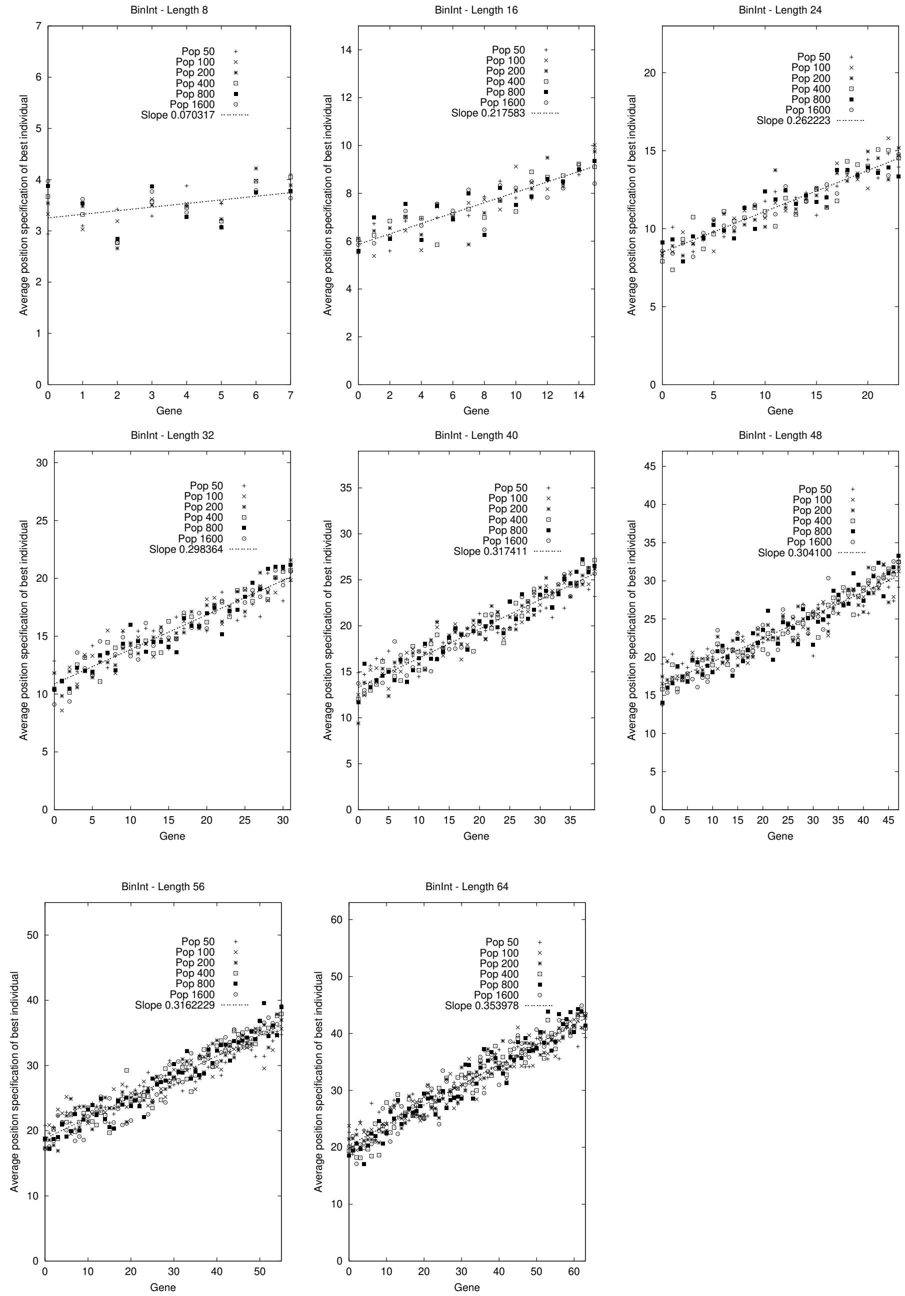


Fig. 2. Results for the InvBinInt problem. The graphs plot the average *locus* specification of each gene in the genotype string, averaged over 100 runs, for different problem lengths.

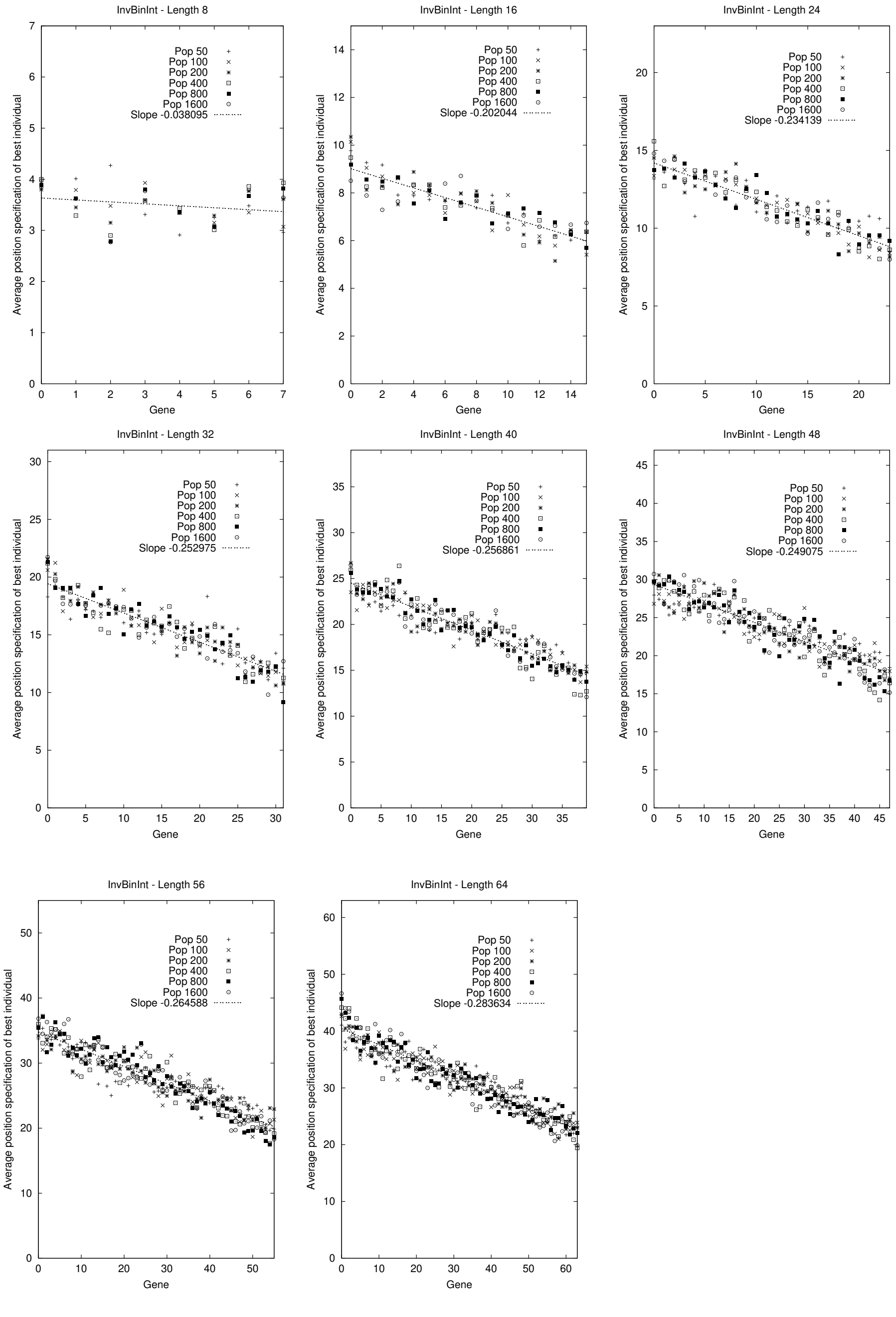
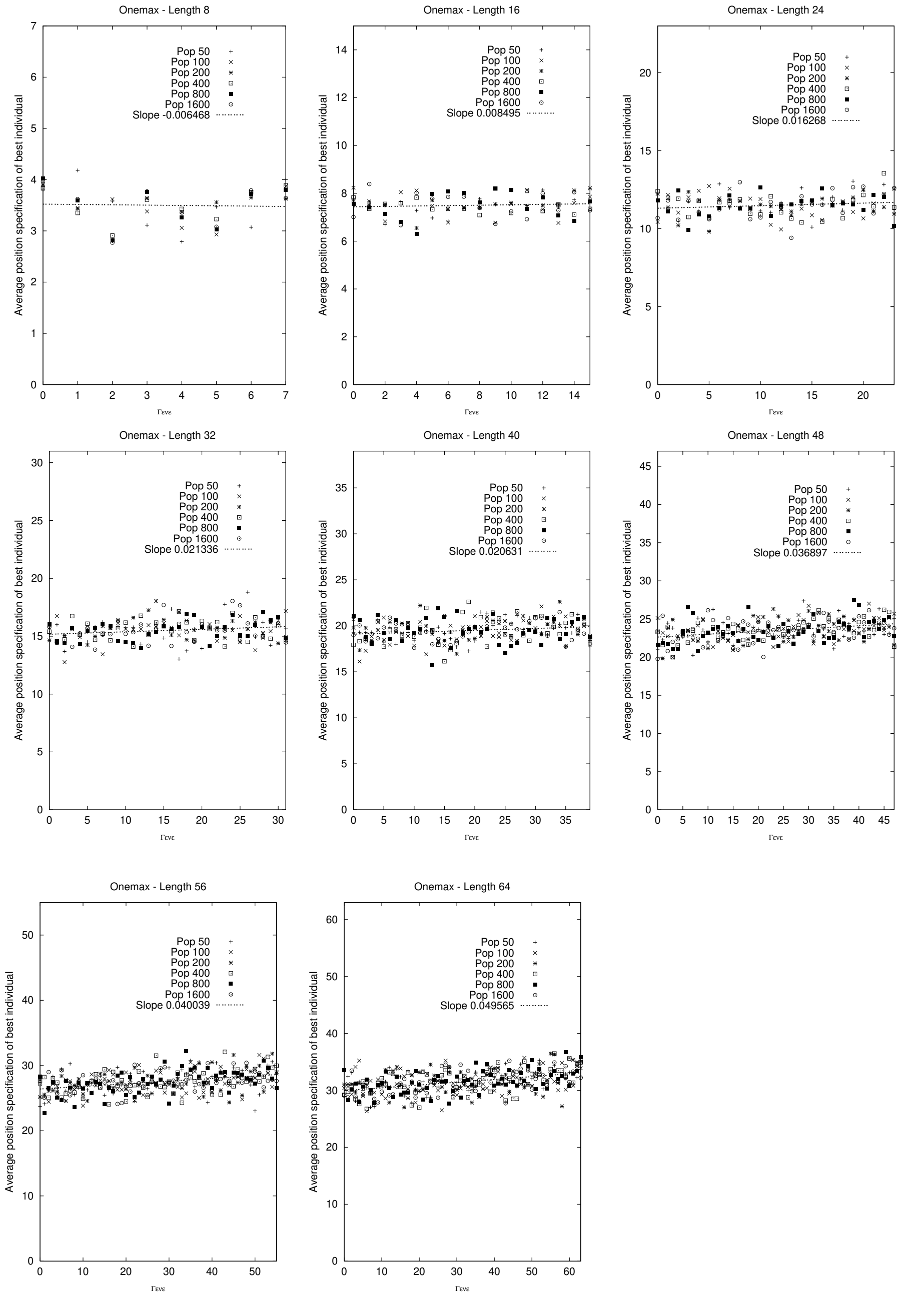


Fig. 3. Results for the Onemax problem. The graphs plot the average *locus* specification of each gene in the genotype string, averaged over 100 runs, for different problem lengths.



By adapting the structure of its genotype strings to match the problem, GAuGE is effectively changing the fitness landscape to facilitate the search. In other words, it is automatically co-evolving its representation, while evolving the solutions for the problem.

It is also interesting to observe that the slope of the regression lines tends to increase with longer strings. This suggests that the use of longer strings (more difficult problems) accentuates the position independent nature of GAuGE, but is also related to the amount of degenerate code used in each problem (as on each problem eight bits were used to encode each position, the amount of degenerate code is smaller on larger problems).

Finally, another use for GAuGE in problems of this nature is to identify the salience structure of specific problems. Stringer and Wu [14] have recently presented an effective and elegant method for this purpose. However, being based on a standard Genetic Algorithm, their method requires much processing, as counts of unique alleles and unique sub-genotypes are required; in GAuGE, only best-of-run individuals are processed, by averaging and plotting them.[2]. Also, they technique requires an absence of mutation for clarity of results, whereas in GAuGE the system is ran *as is*, i.e. no changes to it are required to identify the salience structure of a given problem.

6 Conclusions

This work is an investigation into functional dependency within a genotype string, and how it improves the process of solving problems with scaled salience; it shows that individuals can be built from left to right regardless of the salience hierarchy, and sheds some light onto this process. This helps in understanding not only how GAuGE works, but also GE, and seems to suggest that, contrary to general belief, functional dependency can improve the performance of algorithms where this feature is present.

It has been shown here that, in the problems presented, GAuGE can identify their salience hierarchy, and reorder its genotype string to match it, without any specific mechanism to do so other than its functional dependency. Not only does this helps the system in solving problems with scaled salience, but it also helps in detecting salience in any given problem.

7 Future Work

This work has opened the door to broader research onto the dynamics of GAuGE systems, or even of systems with functional dependency (including GE). Future work involves a more in-depth analysis of the results obtained, such as the different features affecting the slope of the regression lines, and tackling other kinds of problems with scaled salience, such as symbolic regression problems (as used by Stringer and Wu [14]), or problems where building blocks with different salience are used, rather than single genes [6].

[2] Although this is clearly not a precise measure, but rather an estimate.

References

1. Banzhaf, W.: Genotype-Phenotype-Mapping and Neutral Variation - A case study in Genetic Programming. In: Davidor et al., (eds.): Proceedings of the third conference on Parallel Problem Solving from Nature. Lecture Notes in Computer Science, Vol. 866. Springer-Verlag. (1994) 322-332
2. Bean, J.: Genetic Algorithms and Random Keys for Sequencing and Optimization. ORSA Journal on Computing, Vol. **6**, No. 2. (1994) 154-160
3. Harik, G.: Learning Gene Linkage to Efficiently Solve Problems of Bounded Difficulty Using Genetic Algorithms. Doctoral Dissertation, University of Illinois (1997)
4. Keijzer M., Ryan C., O'Neill M., Cattolico M., and Babovic V.: Ripple Crossover in Genetic Programming. In: Miller et al., (eds.): Proceedings of the Fourth European Conference on Genetic Programming. Springer. (pp. 74-86) Lecture Notes in Computer Science, Vol. 2038. Springer-Verlag. (2001) 74-86
5. Kimura, M.: The Neutral Theory of Molecular Evolution. Cambridge University Press. (1983)
6. Lobo, F., Goldberg, D. E., and Pelikan, M.: Time complexity of genetic algorithms on exponentially scaled problems. In: Whitley et al., (eds.): Proceedings of the Genetic and Evolutionary Computation Conference GECCO-2000. Morgan Kaufmann Publishers, San Francisco (2000) 151-158
7. Nicolau, M., and Ryan, C.: LINKGAUGE: Tackling hard deceptive problems with a new linkage learning genetic algorithm. In: Langdon et al., (eds.): Proceedings of the Genetic and Evolutionary Computation Conference GECCO-2002. Morgan Kaufmann Publishers, San Francisco (2002) 488-494
8. O'Neill, M.: Automatic Programming in an Arbitrary Language: Evolving Programs with Grammatical Evolution. Doctoral Dissertation, University of Limerick (2001)
9. O'Neill, M., and Ryan, C.: Grammatical Evolution. IEEE Transactions on Evolutionary Computation, Vol. **5**, No. 4. (2001) 349-358
10. O'Neill, M., and Ryan, C.: Genetic Code Degeneracy: Implications for Grammatical Evolution and Beyond. In: Floreano et al., (eds.): Proceedings of the Fifth European Conference on Artificial Life, ECAL'99. Lecture Notes in Computer Science, Vol. 1674. Springer-Verlag. (1999)
11. Rudnick M. Genetic Algorithms and Fitness Variance with an Application to the Automated Design of Articial Neural Networks. Unpublished Doctoral Dissertation, Oregon Graduate Institute of Science and Technology (1992)
12. Ryan, C., Collins, J.J., and O'Neill, M.: Grammatical Evolution: Evolving Programs for an Arbitrary Language. In: Banzhaf et al., (eds.): Proceedings of the First European Workshop on Genetic Programming, EuroGP'98. Lecture Notes in Computer Science, Vol. 1391. Springer-Verlag. (1998) 83-95
13. Ryan, C., Nicolau, M., and O'Neill, M.: Genetic Algorithms using Grammatical Evolution. In: Foster et al, (eds.): Proceedings of EuroGP-2002. Lecture Notes in Computer Science, Vol. 2278. Springer-Verlag. (2002) 278-287
14. Stringer, H., and Wu, A. S.: A Simple Method for Detecting Domino Convergence and Identifying Salient Genes Within a Genetic Algorithm. In: Langdon et al., (eds.): Proceedings of the Genetic and Evolutionary Computation Conference GECCO-2002. Morgan Kaufmann Publishers, San Francisco (2002) 594-601
15. Thierens, D., Goldberg, D. E., and Pereira, A.G.: Domino convergence, drift, and the temporal-salience structure of problems. In: Proceedings of the 1998 IEEE World Congress on Computational Intelligence. (1998) 535-540