On the use of gene dependency to avoid deceptive traps

Miguel Nicolau
C.S.I.S. Department
University of Limerick
Ireland
Miguel.Nicolau@ul.ie

Conor Ryan
C.S.I.S. Department
University of Limerick
Ireland
Conor.Ryan@ul.ie

Abstract

This paper presents a new approach to the field of genetic algorithms, based on the introduction of dependency between genes, as inspired by Grammatical Evolution. A system based on that approach, *LINKGAUGE*, is presented, and results reported show how the dependency between genes creates a tight linkage, guiding the system to success on hard deceptive linkage problems.

1 INTRODUCTION

Since they were first introduced, genetic algorithms (Holland, 1975; Goldberg, 1989) have been considered good general problem solvers, and have been applied to many problems, showing their flexibility. In the standard approach, an individual consists of a string of values, and operators are provided to combine those values, so that building blocks are constructed and combined to form solutions. However, no mechanism exists to ensure a linkage between the values of those blocks (Goldberg, Deb, Korb, 1991); if using standard genetic operators, this leads to an easy disruption of those blocks, and to an inability to scale-up efficiently.

According to (Goldberg, Deb, Thierens, 1993), a successful algorithm should not only concentrate on the production of building blocks, but also on maintaining and exchanging those blocks between individuals.

In recent years, much work has been done on achieving a tighter linkage between genes, and a family of algorithms called competent GAs has emerged; these have been largely based on the idea encoding both the position and the value of each element of an individual. These algorithms have proven to be successful when applied to hard deceptive linkage problems.

In this paper, we present a new system, LINKGAUGE, which uses the idea of functional dependency between genes to tackle the class of deceptive linkage problems. This system is an extension of GAUGE (Genetic Algorithms Using Grammatical Evolution), a system introduced in (Ryan, Nicolau, O'Neill, 2002) and based on the idea of position independence for the values encoded by each gene; GAUGE, in turn, uses many of the ideas behind Grammatical Evolution (Ryan, Collins, O'Neill, 1998; O'Neill, Ryan, 2001).

Our aim is to investigate the effect of gene dependency when tackling hard deceptive linkage problems. To do so, we apply LINKGAUGE to two problems of this class, and compare it to the original GAUGE system and to the standard messy genetic algorithm (Goldberg, Korb, Deb, 1989). Our results show by comparison that not only the new system solves the problems presented, but it is faster, scales better, and requires less hardware resources than the messyGA.

2 GRAMMATICAL EVOLUTION

Grammatical Evolution (GE) is an evolutionary algorithm approach to automatic program generation, which evolves strings of binary values, and uses a BNF grammar to map these into programs. This mapping involves transforming the binary individual into a string of integer values, and then using those values to choose transformations from the grammar, to map a start symbol into a syntactically correct program.

The main nature-inspired features used in GE are:

- genotype to phenotype mapping;
- degenerate genetic code;
- functional dependency between genes.

The genotype to phenotype mapping process is applied as follows: a string of binary values (genotype)

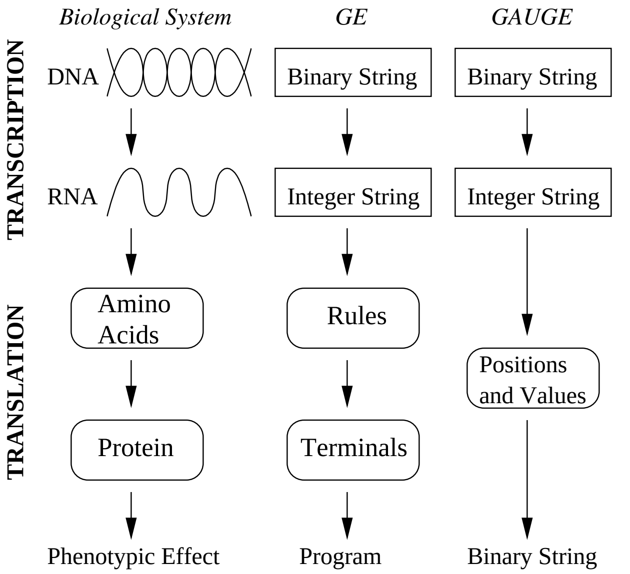

Figure 1: Genotype to Phenotype mapping

is evolved, and, before being evaluated, is subjected to a mapping process to create a program (phenotype), which is then evaluated by the fitness function.

The degenerate genetic code also plays a role in the performance of GE (O'Neill, Ryan, 1999); since the *mod* function is used to map each integer to a finite number of grammar rules, different integers can select the same rule. The genotype can therefore be modified without affecting the phenotype.

Finally, the function of each value in the integer string depends on those preceding it, as they determine which non-terminal symbols remain to be mapped. This creates a linkage between each gene on the chromosome and those that precede it, helping the individual preserve good building blocks during the evolutionary process, when subjected to the harsh effects of genetic operators. This has been termed the "Ripple Effect" (Keijzer et al., 2001).

3 GAUGE

GAUGE is based on many of the same ideas behind the implementation of GE. It too uses a genotype to phenotype mapping process: an individual is composed of a binary sequence (genotype) which, once ready for evaluation, is mapped onto a string of integers, which are decoded as a series of (*position, value*) pairs to finally build a new binary string (phenotype), ready to be evaluated. Figure 1 illustrates this process, and compares it to GE's analogy to molecular biology.

The degenerate genetic code is also seen in GAUGE: as in GE, the integer string is encoded using 8 bits per gene, and the *mod* function maps the resulting value onto the range of accepted values, resulting in different bit combinations encoding the same integer.

When mapping a (*position, value*) pair onto the phenotype string, and to avoid over and under-specification, GAUGE uses the principle of functional dependency between genes: each position reference depends on those preceding it, creating a left side dependency as in GE, and a tight linkage between all genes in each individual. This process is explained in Section 3.1.

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 (Bean, 1994) with the Random Keys Genetic Algorithm (RKGA) hinted that a tight linkage between genes would result in a smoother transition between parents and offspring, therefore respecting the linkage between the values within the genotype.

3.1 EXAMPLE GAUGE MAPPING

Let us take the following binary string as an example:

`0110 0111 0001 0100 0111 1001 0010 0011`

The first step is to map it onto an integer string. For the purpose of clarity, we will use only four bits to encode each integer, and therefore end up with:

`6 7 1 4 7 9 2 3`

This string represents a sequence of four (*position, value*) pairs, and will be used to fill a string of four bits (phenotype). We take the first position, `6`, and *mod* it onto the number of available positions in the final string (i.e., 4), obtaining `6 % 4 = 2` (the third position in the phenotype string). In a similar way, we map the *value* of that pair (7) into a binary value: `7 % 2 = 1`. Finally we place this value on the phenotype string:

`? ? 1 ?`

With 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 (`4 % 2 = 0`):

`? 0 1 ?`

With the pair (7,9), we map the position 7 onto the number of remaining positions: 7 % 2 = 1 (second free position, i.e. the last position in the string); and map 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 (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 string will therefore be:

```
1 0 1 1
```

3.2 EARLY RESULTS

In (Ryan, Nicolau, O'Neill, 2002), GAUGE was applied to a standard genetic algorithm problem and a deceptive ordering problem. On the former, it performed as well as a simple GA, showing that its mapping process does not affect its performance in simple problems; on the latter, its (*position,value*) specification provided a way of swapping elements in a solution, helping the system to avoid local optima. The interested reader is referred to the mentioned paper.

4 LINKGAUGE

The idea behind LINKGAUGE is to extend the gene dependency, seen in GAUGE when mapping positions of the phenotype string, to the mapping of the *contents* of that string. This is done by introducing a dependency of each gene-specified value on the gene-specified values preceding it: when a value is placed on the phenotype string, it is calculated by adding all the previous *value* fields in each (*position, value*) pair, and normalizing the result to the range of accepted values. The value each gene encodes can be calculated by:

$$(\sum_{i=0}^{n} x_i)\%v$$

where
n = order of the gene (i.e. gene 0, gene 1, etc)
x_i = number in *value* field for gene i
v = value to normalize (for binary strings, 2 is used)

It should be noted that, theoretically, any function could be used to introduce this dependency; further research will analyze the suitability of other functions.

4.1 EXAMPLE LINKGAUGE MAPPING

Following the GAUGE mapping example, the pair (6,7) will generate the same string as before:

```
?  ?  1 ?
```

With the next pair, however, the value is calculated by (7+4) % 2 = 1 (the sum of the previous *value* fields mapped to the binary range). The position is the same as before (1 % 3 = 1), so we end up with the string:

```
?  1 1 ?
```

In the next pair, the value is calculated by (7+4+9) % 1, giving the value 0, and the final value is calculated by (7+4+9+3) % 1 = 1. The final string will be:

```
1 1 1 0
```

The objective of this mapping is to create a tight linkage between the values encoded by each gene. The previously mentioned "Ripple Effect" is therefore extended to the values within the genes themselves.

5 EXPERIMENTAL RESULTS

We tested our system with an order-three deceptive problem and an order-five deceptive problem (Goldberg, Deb, Kargupta, Harik, 1993), with string lengths of 30 through 105 bits, and compared our system to the original GAUGE system and the standard messyGA.

The original GAUGE system failed to find a solution for any of the two problems; this shows that the extension of gene dependency to the specification of both positions and values (as in LINKGAUGE) is crucial to the performance of the system on deceptive linkage problems. We therefore analyze the results obtained with both LINKGAUGE and the messyGA (Figure 2).

Results for the order-three problem show the messyGA performing better with small string lengths, but losing performance as the problem gets harder, failing to find a solution with strings of 60-bits or more. LINKGAUGE, however, looses little performance as the problem gets harder, showing its ability to scale-up to an increasing problem difficulty.

In the order-five problem, the messyGA failed to find a solution over a maximum set number of function calls, due to the difficult nature of the problem. The results obtained with LINKGAUGE were, however, similar to those obtained in the order-three problem, suggesting that the extra deceptiveness of order-five problems doesn't have as strong an impact on its performance, thus underlining the scale-up properties of the system.

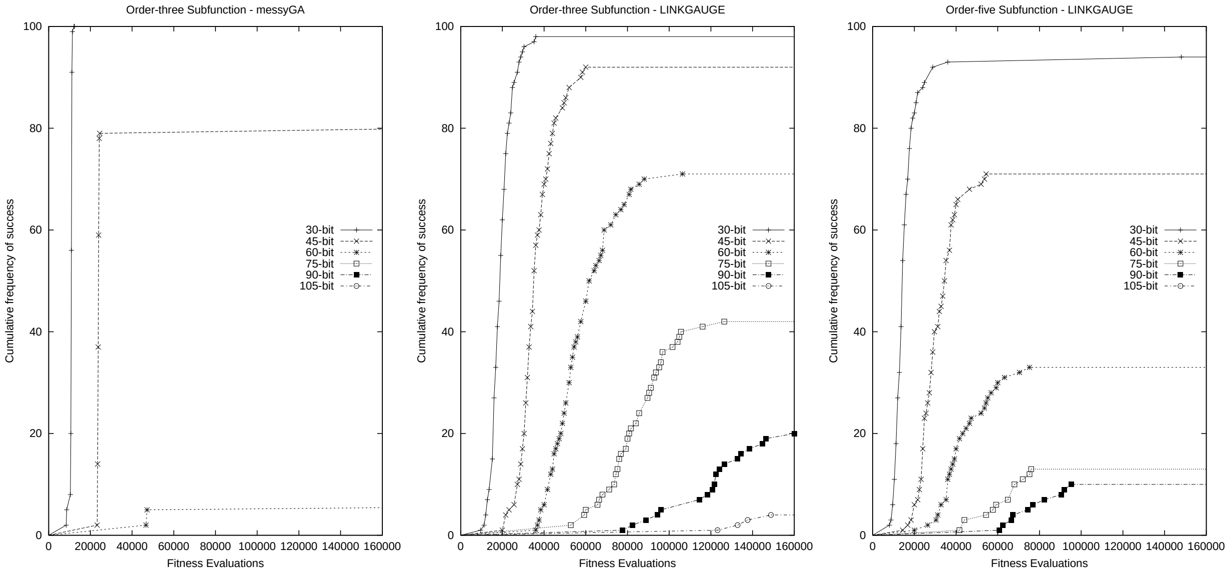

Figure 2: Order-three results for messyGA (left) and LINKGAUGE (center); the former failed to find any solutions with string lengths of over 60 bits, whereas the latter is shown to scale better when using longer strings. That scalability is further illustrated with the results obtained for the order-five problem (right).

6 CONCLUSIONS AND FUTURE WORK

The results obtained show how, by introducing dependency between the values encoded by each gene on the GAUGE system, a tight linkage between the values of the genotype string is created, which helps the system avoid hard deceptive linkage traps.

Future work will involve a close examination of the effects of gene dependency on the evolution process of LINKGAUGE, and how the construction of building blocks is affected by this technique.

References

Bean, J. 1994. Genetic Algorithms and Random Keys for Sequencing and Optimization. ORSA Journal on Computing, Vol. 6, No. 2, Spring 1994. (pp. 154-160)

Goldberg, D. E. 1989. Genetic Algorithms in Search, Optimization and Machine Learning. Addison Wesley.

Goldberg, D. E., Deb, K., Kargupta, H., and Harik, G. 1993. Rapid, Accurate Optimization of Difficult Problems Using Fast Messy Genetic Algorithms. Illinois Genetic Algorithms Laboratory, report no. 93004.

Goldberg, D. E., Deb, K, and Korb, B. 1991. Don't Worry, be Messy. In *Proceedings of ICGA'91*, R. Belew and L. Booker, Eds., Morgan Kaufman. (pp. 24-30)

Goldberg, D. E., Deb, K., and Thierens, D. 1993. Toward a Better Understanding of Mixing in Genetic Algorithms. Journal of the Society of Instrument and Control Engineers, Vol. 32, No. 1. (pp. 10-16)

Goldberg, D. E., Korb, B., and Deb, K. 1989. Messy genetic algorithms: Motivation, analysis, and first results. in *Complex Systems, 3.* (pp. 493-530)

Holland, J. 1975. Adaptation in Natural and Artificial Systems. Ann Arbor, University of Michigan Press.

Keijzer M., Ryan C., O'Neill M., Cattolico M., and Babovic V. 2001. Ripple Crossover in Genetic Programming. In *LNCS 2038, Proceedings of EuroGP 2001*, Springer. (pp. 74-86)

O'Neill, M., and Ryan, C. 1999. Genetic Code Degeneracy: Implications for Grammatical Evolution and Beyond. In *ECAL'99: Proceedings of the Fifth European Conference on Artificial Life.*

O'Neill, M., and Ryan, C. 2001. Grammatical Evolution. IEEE Transactions on Evolutionary Computation, Vol. 5, No. 4. (pp. 349-358)

Ryan, C., Collins, J.J., and O'Neill, M. 1998. Grammatical Evolution: Evolving Programs for an Arbitrary Language. In *LNCS 1391, Proceedings of EuroGP 98*, Springer-Verlag. (pp. 83-95)

Ryan, C., Nicolau, M., and O'Neill, M. 2002. Genetic Algorithms using Grammatical Evolution. In *LNCS 2278, Proceedings of EuroGP 2002*, Springer-Verlag. (pp. 279-288)