LINKGAUGE: Tackling hard deceptive problems with a new linkage learning genetic algorithm

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

A novel approach to obtaining a tight linkage between genes in a genetic algorithm is described, and a new system based on that approach, LINKGAUGE, is proposed. Experiments presented draw a comparison between the standard messy genetic algorithm and LINKGAUGE, and show that the latter avoids deceptive traps and early convergence, with minimal computational cost. The scalability potential of the new approach is illustrated with results for two hard deceptive problems.

1 INTRODUCTION

Since they were first introduced, genetic algorithms (Holland, 1975; Goldberg, 1989) have been considered good all-round general problem solvers, and have since been applied to a variety of problems, which show their flexibility and adaptability. In the standard approach, each individual consists of a sequence of values, and operators are provided to exchange and combine those values, so that building blocks (short, highly fit sequences of values) are constructed, and later combined to form correct solutions. However, there is no mechanism to ensure a tight linkage between the values of those sequences (Goldberg, Deb, Korb, 1991); when applying standard genetic operators, this leads to an easy disruption of building blocks, rather than their maintenance (Harik, 1997), and therefore to an inability to scale-up to more difficult problems. Furthermore, an individual's genes are position dependent, in that a given locus on the genome always codes for the corresponding bit position in the phenotype. This can make crossover even less likely to maintain useful building blocks, especially if they represent geographically distant positions in the phenotype.

According to (Goldberg, Deb, Thierens, 1993), a successful algorithm should not only concentrate on the production of building blocks, but also on their preservation and exchange 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 (Goldberg, 2001); these are mostly based on the idea of genes coding both the position and the value of each element of an individual. These algorithms have proven to be successful when applied to hard problems, such as deceptive linkage problems (Goldberg, Korb, Deb, 1989; Goldberg, Deb, Kargupta, Harik, 1993; Harik, 1997; Pelikan, Goldberg, Cantú-Paz, 1999).

In this paper, we present a new system, LINKGAUGE, which tackles the class of deceptive linkage problems by using a simple yet effective algorithm. This system is an extension of GAUGE (Genetic Algorithms Using Grammatical Evolution), a system described in (Ryan, Nicolau, O'Neill, 2002) and based on the idea of encoding a position/value couple on each gene, to create a position-independent algorithm; GAUGE, in turn, employs many of the ideas behind Grammatical Evolution (Ryan, Collins, O'Neill, 1998; O'Neill, Ryan, 2001; O'Neill, 2001). So far, GAUGE has been successfully applied to both standard and deceptive ordering problems.

Our aim when running the experiments described in this paper was to test the aptitude of LINKGAUGE to solve linkage problems, and its scalability when presented with more difficult problems; to do so, we applied the system to two hard deceptive linkage problems, and compare its performance to the standard messy Genetic Algorithm (Deb, Goldberg, 1991). By extending GAUGE's mapping mechanism, we have built a new flexible approach to this kind of problem; our results show by comparison that it finds a solution faster, scales better to harder versions of the problem, and requires far less hardware resources than the messy GA[1].

This paper is organized as follows: we start by briefly introducing Grammatical Evolution in section 2, followed by an explanation of how GAUGE works (section 3) and its extension into LINKGAUGE (section 4). In section 5 we present the problems used for our experiments, and in section 6 we present our results. Finally an analysis of those results is made and conclusions are drawn in section 7, followed by the outline of some future directions of research in section 8.

2 GRAMMATICAL EVOLUTION

GAUGE is based upon many of the techniques implemented in Grammatical Evolution, so we start with an introduction to this system, to highlight the similarities and differences between the two systems.

Grammatical Evolution (GE) is an evolutionary algorithm approach to automatic program generation, which evolves strings of binary values, and uses a BNF (Backus-Naur Form) grammar to map the strings 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 given grammar, so that a start symbol is mapped into a syntactically correct program.

This process is based on the idea of a genotype to phenotype mapping: an individual comprised of binary values (genotype) 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. This creates two distinct spaces, a search space and a solution space.

The degenerate genetic code employed in GE also plays a role in the performance of the system, as seen in (O'Neill, Ryan, 1999); by using the *mod* function to normalize each integer to a finite number of production rules, different integer values can be used to select the same rule. The genotype can therefore be modified without necessarily affecting the phenotype, in a process known as neutral mutations (Kimura, 1983; Banzhaf, 1994).

Finally, the functionality of the values in the integer string is dependent on the values preceding it, as those determine which non-terminal symbols remain to be mapped. This creates a linkage between each gene on the chromosome and all those which precede it, and helps the individual in preserving good building blocks during the evolution process, where it is subjected to the harsh effects of operators like crossover. This has been termed the "Ripple Effect" (Keijzer, Ryan, O'Neill, Cattolico, Babovic, 2001).

[1]Due to its variable length nature, some messy GA runs required over 1GB of memory to store a population, comparing to less than 1MB for the most demanding LINKGAUGE runs.

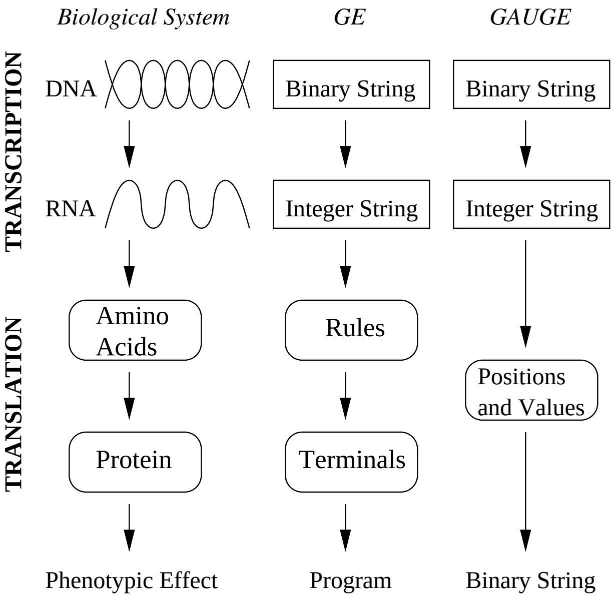

Figure 1: Genotype to Phenotype mapping

3 GAUGE

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

Another feature of GE upon which GAUGE is based is that the function of a gene in an individual depends on the value of the genes preceding it; this creates a tight linkage between adjacent genes in that individual.

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, in a process we call competitive building blocks.

Work by Bean (Bean, 1994) 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.

3.1 EXAMPLE GAUGE MAPPING

In this subsection we take a look at how an individual is created and evaluated using GAUGE. Let us take as an example individual the following binary sequence:

```
0110 0111 0001 0100 0111 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 standard eight used in the actual GAUGE code), and therefore end up with:

```
6 7 1 4 7 9 2 3
```

This string will be evaluated as a sequence of four (*position, value*) pairs, and will be used to fill in a string of four bits. We therefore take the first position, `6`, and map it onto the number of available positions in the final string (i.e., `4`), by calculating the remainder of the division of `6` by `4` (`6 % 4`), giving the value `2` (i.e., the third position in the phenotype string). We use the same mapping process to transform the value for that position, `7`, into a binary value: `7 % 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, (`7,9`), we map the position `7` onto the number of available positions, `2`, by calculating `7 % 2 = 1` (second free position, which is the last 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 EARLY RESULTS

In (Ryan, Nicolau, O'Neill, 2002), GAUGE was applied to both a standard genetic algorithm problem and a 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 reflect in a loss of performance in simple problems, while on the latter, its (*position,value*) specification was shown to provide the flexibility of swapping elements in a solution, helping the system to avoid local optima. The interested reader is referred to the mentioned paper.

4 LINKGAUGE

In this section the LINKGAUGE system is presented. The idea is to extend the tight linkage between the gene positions, as seen in GAUGE, to the gene values themselves. This is achieved by extending GAUGE's mapping process: every time a value is to be placed on the phenotype string, it is calculated by adding all the previous *value* fields in each (*position, value*) pair and then normalizing the result over the range of accepted values. The value each gene will provide can therefore be calculated by the formula

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

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 dependency between the values; the suitability of other functions will be the subject of further research.

4.1 EXAMPLE LINKGAUGE MAPPING

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

```
?  ?  1 ?
```

In the next pair, however, the value is calculated by `(7+4) % 2` (i.e., the cumulative total of the previous *value* fields normalized over the range of binary numbers), giving the value `1`. The position calculation is the same as before (`1 % 3 = 1`), so we end up with the string:

```
?  1 1 ?
```

In the next pair, the value will be 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 value of the genes. The previously mentioned "Ripple Effect" is therefore extended to the values within the genes themselves.

5 DECEPTIVE PROBLEMS

In this section we introduce the two deceptive problems which we used on our experiments. These were used to test the performance of LINKGAUGE, and to compare it to the messy GA, using the mGA code available in the IlliGAL web site and described in (Deb, Goldberg, 1991). We chose to compare our system to the messy GA as the latter is the origin of most modern competent GAs, introducing the concepts of primordial and juxtapositional phases, over- and under-specification, and competitive templates. Future work should include comparisons to other more recent competent GAs.

5.1 ORDER-THREE DECEPTIVE PROBLEM

The order-three deceptive problem was the first problem reported using the original mGA, in (Goldberg, Korb, Deb, 1989). In the original problem, ten order-three deceptive sub-functions are concatenated together to form a 30-bit length problem. We have extended the problem, and used lengths of 30, 45, 60, 75, 90 and 105 bits.

Each sub-function has a global optimum (`000`) and a deceptive local optimum (`111`). The objective is to create a series of local optima that will attempt to keep the systems from reaching the one and only global optimum; on the 105-bit problem, this means there are 2^{105} (`4.05e+31`) possible solutions, with 2^{35} (`3.44e+10`) optima (local and global optimum combinations within each of the sub-functions), of which only one is the global solution for the entire string. Table 1 shows the function values for every 3-bit combination.

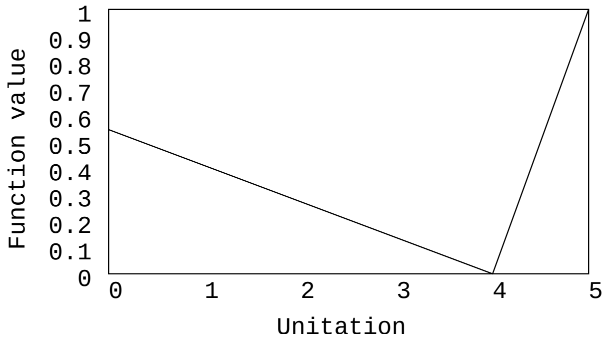

Figure 2: Order-Five Deceptive Problem Unitation Graph.

Table 1: Order-Three Sub-function Values.

String	Value	String	Value
000	28	100	14
001	26	101	0
010	22	110	0
011	0	111	30

5.2 ORDER-FIVE DECEPTIVE PROBLEM

In (Goldberg, Deb, Kargupta, Harik, 1993), a performance comparison between the original messy GA and the Fast Messy Genetic Algorithm is made, by using both the order-three sub-function and an order-five sub-function; we used the same problems in our tests.

In this problem, substrings of five bits are concatenated together, with the global optimum being (`11111`) and the local optimum (`00000`). Figure 2 shows this problem in terms of a unitation graph, i.e. the number of `1`s in a sub-function determines its fitness. This function is fully deceptive, as can be seen in (Deb, Goldberg, 1994).

6 EXPERIMENTS

In this section we present the results obtained on the two described problems, using both LINKGAUGE and the original messy GA. We start by describing the experimental setup used on each system, follow with an overview of the results obtained in our experiments, and conclude this section with a discussion of those results.

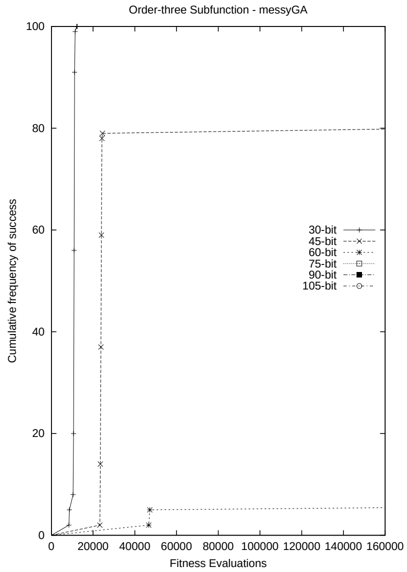

Figure 3: Order-Three Results For MessyGA; No Results Were Obtained With String Lengths Over 60 Bits.

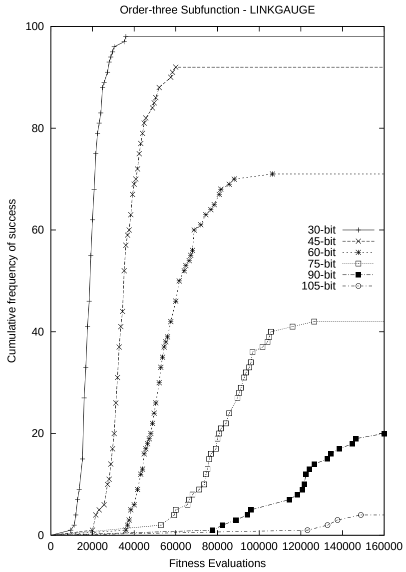

Figure 4: Order-Three Results For LINKGAUGE.

6.1 EXPERIMENTAL SETUP

We used a standard configuration with LINKGAUGE for this problem. With a population of 800 individuals, the replacement strategy used was steady-state, and the selection routine was roulette-wheel; probability of crossover was set to 0.9, and mutation was set to 0.01, which are the standard Genetic Programming (Koza, 1992) values used in GE; no attempt was made to optimize these values. The maximum number of fitness function calls was set to `1.6e+04`, on both systems.

Table 2: Tested combinations of settings for the messy GA algorithm.

Parameters	Set 1	Set 2	Set 3	Set 4
Maximum era	3	3	4	3
Prob. cut	0.02	0.02	0.02	0.02
Prob. splice	1.0	1.0	1.0	1.0
Prob. allelic mut.	0.0	0.0	0.0	0.0
Prob. genic mut.	0.0	0.0	0.0	0.0
Thresholding	no	yes	yes	yes
Tie-breaking	no	yes	yes	yes
Reduced popsize	no	yes	yes	yes
Extra members	no	no	no	no
Copies	5,1,1	5,1,1	5,1,1,1	5,1,1
Total generations	20	20	15	100
Juxtapos. popsize	250	250	250	100

In the messy GA, a range of different sets of parameters were tried as seen in Table 2; a detailed explanation of these settings can be found in (Goldberg, Korb, Deb, 1989; Goldberg, Deb, Korb, 1991). Settings 1 (the standard messy GA values) and 2 behaved well with the 30-bit string problem, but gave very poor results with longer string lengths, and were therefore discarded; settings 3 and 4 gave the improved results, with setting 3 achieving the best performance on the harder problems, and so was chosen for our comparison.

6.2 RESULTS

Both systems were applied to each of the problems over 100 runs, and the graphs presented here show the cumulative number of successful runs plotted against the number of fitness evaluations required.

6.2.1 Order-Three Problem

Results for the order-three problem, shown in Figures 3 and 4, show the messy GA achieving a superior performance with small string lengths. However, as the string length gets longer, it can be seen that LINKGAUGE scales better to the problem; with lengths of 75-bit, 90-bit and 105-bit, all messy GA runs failed to find one instance of the global optimum (a string composed of all `1s`), and therefore they are not plotted in the graph presented.

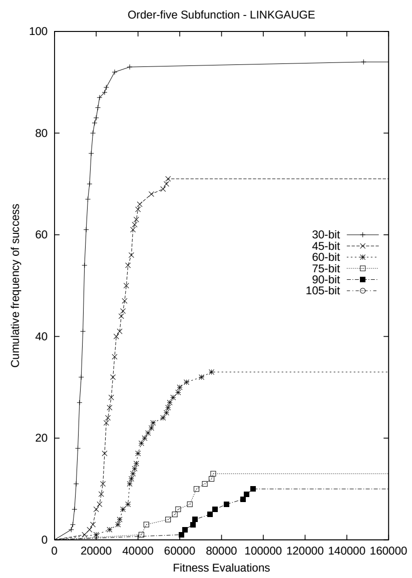

Figure 5: Order-Five Results for LINKGAUGE.

6.2.2 Order-Five Problem

According to results published in (Goldberg, Deb, Kargupta, Harik, 1993), the original messy GA with standard parameter settings would need a number close to `1e+08` function evaluations to find an instance of a global solution for the order-five problem, with a 50-bit string. It is therefore no wonder that in our tests, the messy GA failed to find any instance of the global solution over `1.6e+04` fitness function calls[2].

The results obtained with LINKGAUGE (shown in Figure 5) are, however, similar to those obtained in the order-three problem, which suggests that the extra deceptiveness of order-five problems doesn't have as strong an impact on its performance as one might expect. Over the allowed number of fitness evaluations, only in the 105-bit problem did LINKGAUGE not find any solution. It should be mentioned, however, that in this class of problem LINKGAUGE worked better with a population size of 1600 individuals using the same number of fitness function calls (and indeed found solutions for the 105-bit problem), which tends to suggest that a better trade-off between number of generations versus population size can be found; this could, however, be looked upon as parameter optimization, and therefore those results are not reported here in detail.

[2]It also increased its hardware requirements exponentially; on one specific run, to specify the contents of a 30-bit string, the average length of an individual was over 22000 genes.

6.3 ANALYSIS

On the standard order-three problem, a direct comparison with the messy GA shows LINKGAUGE's ability to adapt to an increasing problem difficulty; although with smaller strings the messy GA is faster at finding a solution (with any of the parameter sets tested), it does not present the scalability of LINKGAUGE when the problem gets harder.

On the order-five deceptive problem, the lack of results for the messy GA, and the only slight loss of performance of LINKGAUGE, underline the scale-up properties of the latter. Results obtained (but not reported, for the sake of clarity) have shown that with larger population sizes and the same number of fitness evaluations, LINKGAUGE's performance in this problem increased, which leads to some optimism as to the system's ability to avoid early convergence.

7 CONCLUSIONS

We have presented a new genetic algorithm based system, LINKGAUGE, for the purpose of solving hard deceptive linkage problems. The results reported show an interesting scale-up property for our system, which is remarkable given that it is based on a simple genetic algorithm; no specific genetic operators have been introduced, and parameters such as crossover and mutation rate have been set to standard values. This is not the case of the system compared to, messy GA, which has a specific implementation that slightly diverges from the original implementation ideas of genetic algorithms: although a good approach in itself, this does make the algorithm harder to use and understand, and parameter tuning was required to achieve a good performance. It should also be mentioned that LINKGAUGE's fixed-length, fixed-population size nature results in an algorithm that has very little hardware and over-head processing requirements, especially when compared to the original messy GA.

8 FUTURE WORK

Future lines of research include numerical and statistical analysis of the data presented, to effectively measure the performance and the degree of scalability of the system. Also, a rigorous comparison should be made between the system presented and other more recent linkage learning genetic algorithms (Goldberg, Deb, Kargupta, Harik, 1993; Harik, 1997), to highlight similarities and differences, and advantages/disadvantages.

Acknowledgments

The authors would like to thank Dr. Michael O'Neill for his advice on writing this paper, and for his driving force in the initial implementations of GAUGE.

References

Banzhaf, W. 1994. Genotype-Phenotype-Mapping and Neutral Variation - A case study in Genetic Programming. In *Parallel Problem Solving from Nature III*, Springler. (pp. 322-332)

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)

Deb, K., and Goldberg, D. E. 1991. mGA in C: A Messy Genetic Algorithm in C. Illinois Genetic Algorithms Laboratory (IlliGAL), report no. 91008.

Deb, K., and Goldberg, D. E. 1994. Sufficient Conditions for Deceptive and easy Binary Functions. *Annals of Mathematics and Artificial Intelligence 10.* (pp. 385-408)

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

Goldberg, D. E. 2001. The Design of Competent GAs: Toward a Computational Theory of Innovation. Tutorial presented at the Genetic and Evolutionary Computation Conference, San Francisco, July 2001.

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 (IlliGAL), report no. 93004.

Goldberg, D. E., Deb, K, and Korb, B. 1991. Don't Worry, be Messy. In *Proceedings of the Fourth International Conference on Genetic Algorithms (San Mateo, CA)*, 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)

Harik, G. 1997. Learning Gene Linkage to Efficiently Solve Problems of Bounded Difficulty Using Genetic Algorithms. Doctoral Dissertation, University of Michigan, Ann Arbor.

Holland, J. H. 1975. Adaptation in Natural and Artificial Systems. Ann Arbor, MI: 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 the Fourth European Conference on Genetic Programming*, Springer. (pp. 74-86)

Kimura, M. 1983. The Neutral Theory of Molecular Evolution. Cambridge University Press.

Koza, J. 1992. *Genetic Programming.* MIT Press.

O'Neill, M. 2001. Automatic Programming in an Arbitrary Language: Evolving Programs with Grammatical Evolution. Doctoral Dissertation, University of Limerick.

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)

Pelikan, M., Goldberg, D. E., and Cantú-Paz, E. 1999. BOA: The Bayesian Optimization Algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference GECCO-99*, Morgan Kaufman. (pp. 525-532)

Ryan, C., Collins, J.J., and O'Neill, M. 1998. Grammatical Evolution: Evolving Programs for an Arbitrary Language. In *LNCS 1391, Proceedings of the First European Workshop on Genetic Programming*, Springer-Verlag. (pp. 83-95)

Ryan, C., Nicolau, M., and O'Neill, M. 2002. Genetic Algorithms using Grammatical Evolution. In *Proceedings of EuroGP-2002.* (to appear)