# Genetic Algorithms Using Grammatical Evolution

Conor Ryan, Miguel Nicolau, and Michael O'Neill

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

**Abstract.** This paper describes the *GAUGE* system, Genetic Algorithms Using Grammatical Evolution. GAUGE is a position independent Genetic Algorithm that uses Grammatical Evolution with an attribute grammar to dictate what position a gene codes for. GAUGE suffers from neither under-specification nor over-specification, is guaranteed to produce syntactically correct individuals, and does not require any repair after the application of genetic operators.
GAUGE is applied to the standard onemax problem, with results showing that its genotype to phenotype mapping and position independence nature do not affect its performance as a normal genetic algorithm. A new problem is also presented, a deceptive version of the Mastermind game, and we show that GAUGE possesses the position independence characteristics it claims, and outperforms several genetic algorithms, including the competent genetic algorithm messyGA.

## 1 Introduction

Since the inception of the field, Genetic Algorithms[5] [3] have been hamstrung by the dogma that each locus on a genome codes for a particular trait, and that the locus-trait relationship holds not only across an entire population, but even across otherwise independent runs on a problem.

Recently, this has been acknowledged to be a problem. Individuals are subjected to all manner of disruption at the whims of crossover and selection, neither of which respect the geographical location of loci that may be essential for good fitness. In particular, geographically disparate loci on a chromosome are easily split up, often making it difficult to build on previously discovered schema, and, in some cases, difficult even to maintain current useful schema.

In nature, function is rarely dependent on location. A gene (or, more properly, a gene sequence or *protein*) usually functions independently of its location. Under- and over-specification do not appear to be issues in nature, presumably due to the manner in which phenotypes are mapped from genotypes, which ensures that the necessary genes are expressed at crucial times. Individuals that do not exhibit this talent are often spontaneously aborted, or doomed to a lifetime under the spectre of a debilitating disease.

This paper describes a new *Epigenetic* Algorithm (i.e. that employs a genotype to phenotype mapping) that overcomes the position dependence issue. This is achieved by extending Grammatical Evolution[12] [11] with an attribute grammar that makes the system behave as a position-independent Genetic Algorithm. This system, GAUGE (Genetic Algorithms Using Grammatical Evolution) is far less susceptible to the ravages often associated with crossover, as the system can automatically move important genes closer together as a run progresses.

Our main aim in this paper is to achieve a proof of concept. We start by briefly introducing Grammatical Evolution in section 2, and the implementation ideas behind GAUGE in section 3. We then present two problems in section 4, Onemax and Mastermind, and analyse the performance of GAUGE and other systems on those problems. We discuss the value of our results on section 5, and finally draw our conclusions and future directions of research in section 6.

## 2 Grammatical Evolution

Grammatical Evolution (GE) is an evolutionary automatic programming type system, that uses a combination of a variable length binary string genome and a BNF (Backus Naur Form) grammar to evolve interesting structures.

GE uses a chromosome of numbers encoded using eight bits (termed codons) to dictate which rule from the BNF grammar to apply at each state of the derivation sequence, starting from a defined *start* symbol. While one or more non-terminals symbols exist in a sentence, codons are read from the chromosome and used to govern which of the relevant production rules to apply. GE is fully described in [12][11].

## 3 GAUGE - Building a GA with GE

GE has been applied to all manner of automatic programming problems, from symbolic regression, to C programs, or generation of graphical objects. What all these application areas have in common is that the system is given a collection of *terminals*, the items that can appear in a legal sentence, and associated production rules, that govern how they can be combined.

The common view of Genetic Programming[9] is that, given a particular problem statement, a program that satisfies the fitness function is to be generated. In other words, *given a set of terminals, how should they be arranged so that the fitness function is satisfied?*.

The approach GAUGE takes is that *all* problems, even those traditionally considered the topic of Genetic Algorithms, can be looked upon as automatic programming ones. In particular, if one considers a problem to have two parts, that is, not only *what values should the genes have?* but also *where should the genes reside?*, it is reasonable to compare this class of problems with those above.

When applying GAUGE to a Genetic Algorithm problem, one uses a set of codon pairs, one for each gene position in the original problem. An individual

is processed in a similar manner to GE, by `mod`ing each gene by an appropriate value to produce a useful value. The codons are 8-bit values, giving a degenerate (redundant) encoding. For GAUGE, the values required are the position that the pair will code for, and the value for that bit position. To produce these values, a list of *unspecified* positions is maintained, that is, those positions that have not been given a value yet. Thus, even when evolving individuals that are binary strings, GAUGE always has a distinct genotype and phenotype. The mapping happens in two stages; first a set of (position, value) pairs are generated using the grammar below, while the second step simply puts them in the correct order.

GAUGE operates using a tuple $< G, A, q, l >$ where $G$ is the genome, $A$ the attribute grammar below, $q$ a list of free *locations* in the phenotype and $l$ is the length of the phenotype. The maximum value that can appear in the phenotype is governed by the variable $max$, while the function $rember(n, q)$ returns the $n$th member from list $q$ and removes it from the list.

1. $<\text{start}_n>$ ::= if $(n < l)$ { $<\text{P}_n><\text{V}_n><\text{start}_{n+1}>$}
2. $<\text{P}_n><\text{V}_n>$::= $<\text{Q}_{(G[2n]\%(l-n))}>$ G[2n+1]%max.
3. $<\text{Q}_n>$ ::= rember (n,q)

### 3.1 Example Individual

Consider the following individual, expressed in decimal for brevity:
$G$=**31 17 23 16 237 56**.

We start with the following values, $l$=3, $q$=(0,1,2), $max$=2 and $G$ as above. The grammar is invoked with a call to $<\text{start}_0>$, which is mapped to:
$< P_0 >< V_0 ><\text{start}_1>$

We then match the leftmost part to rule #2 above, which expands to:
$(<\text{Q}_{(G[2*0]\%(3-0))}>$, G[2*0+1]%2) $<\text{start}_1>$

which is $(<\text{Q}_{(G[0]\%3)}>$, G[1]%2) $<\text{start}_1>$ which means we read from the genome vector to get the values as follows: $(<\text{Q}_{(31\%3)}>$, 17%2) $<\text{start}_1>$
The next call is to $Q$, in the form $Q_1$, which returns item index #1 from the queue $q$, which is (0, 1, 2). This causes the value 1 to be returned, and removed from the queue. The expression is now: (1,1)$<\text{start}_1>$
We repeat the above steps with the next non-terminal, that is, $<\text{start}_1>$, which is expanded as in the first step to (1,1)$< P_1 >< V_1 ><\text{start}_2>$. Expanding the $< P_1 >< V_1 >$, we need to read positions 2 and 3 from the genome, to give: (1,1)$(<\text{Q}_{(23\%2)}>$, 16%2) $<\text{start}_2>$
This generates the non-terminal $Q_1$ (notice that the value **23** was moded by 2, because $n$ has increased) so again we return the item in the list with the index 1. The current form of that list is (0,2), so 2 is returned and removed from the list, yielding: (1,1)(2,0)$<\text{start}_2>$. The final step involves mapping the remaining positions, and gives **(1,1)(2,0)(0,0)**. The final step in the mapping is to simply reorder these pairs as dictated by the first number in each pair.

## 4 Problems

This section introduces two problems to which standard GAs, GAUGE and messyGA will be applied. To show that GAUGE is a capable system, we used the standard onemax problem; to illustrate its position-independent nature, we have devised a deceptive ordering problem, based on the Mastermind[1] game.

### 4.1 Onemax

We ran the standard onemax problem (i.e. where the fitness of an individual is the sum of the bits which are equal to 1) using lengths of 50, 100 and 150 bits per individual. To choose which GA to compare GAUGE to, we measured the performance of several systems, shown in Table 1.

We included GAs using 8 bits per gene in this comparison, as they use the same redundant encoding as GAUGE. We ran these systems up to 25000 evaluations, using population sizes of 50, 100, 200, 400, 800 and 1600 individuals. We found that the generational algorithms could not solve the problem with these parameters, but steady-state algorithms performed well (with no significant difference between 1 or 8 bits per codon). We therefore compare GAUGE to a simple GA with steady-state, using 1 bit per codon (setting SGAss).

We also compared GAUGE with messyGA[4] (using the code available from the IlliGAL server[6], explained in [2]), and tried several combinations of settings for it (check Table 2), choosing the one that best performed on this given problem (setting STD). We wanted to test all systems with string lengths of up to 400 bits, but ran into difficulties, since some of the messyGA runs (for length 150) demanded over 1.1GB of memory (i.e. over one thousand times more than GAUGE or the simple GA) to execute, due to messyGA's variable length individuals and variable population sizes; this made running the experiments a very long task, and using longer strings was physically impossible[2].

| Parameters | SGA | SGAss | GA | GAss | GAUGE |
|---|---|---|---|---|---|
| Replacement strategy | gener. | s-state | gener. | s-state | s-state |
| Selection routine | r-wheel | r-wheel | r-wheel | r-wheel | r-wheel |
| Bits per gene | 1 | 1 | 8 | 8 | 8+8 |
| Number of runs | 100 | 100 | 100 | 100 | 100 |
| Probability of crossover | 0.9 | 0.9 | 0.9 | 0.9 | 0.9 |
| Probability of mutation | 0.01 | 0.01 | 0.01 | 0.01 | 0.01 |

**Table 1.** General settings used in all GAs and GAUGE.

[1] Mastermind is a registered trademark of Pressman Toy Corporation, by agreement with Invicta Toys and Games, Ltd., UK.

[2] These experiments were run on a dual-processor Pentium III 1GHz computer, with 2GB of shared memory available.

| Parameters | STD | OPT1 | OPT2 | OPT3 |
|---|---|---|---|---|
| Maximum era | 3 | 3 | 3 | 3 |
| Probability of cut | 0.02 | 0.02 | 0.02 | 0.02 |
| Probability of splice | 1.0 | 1.0 | 1.0 | 1.0 |
| Probability of allelic mut. | 0.00 | 0.01 | 0.01 | 0.01 |
| Probability of genic mut. | 0.00 | 0.01 | 0.01 | 0.01 |
| Thresholding | 0 | 0 | 0 | 1 |
| Tiebreaking | 0 | 0 | 0 | 1 |
| Reduced initial pop. | 1 | 0 | 0 | 0 |
| Extra pop. members | 0 | 0 | 0 | 0 |
| Copies | 10, 1, 1 | 1, 1, 1 | 10, 1, 1 | 10, 1, 1 |
| Total generations | 100, 100, 100 | 100, 100, 100 | 100, 100, 100 | 100, 100, 100 |
| Juxtapositional popsize | 100, 100, 100 | 100, 100, 100 | 100, 100, 100 | 100, 100, 100 |

**Table 2.** Tested combinations of settings for messyGA algorithm [2] [4].

**Results** We can see in Fig. 1 that GAUGE has a similar behaviour to the simple GA, across all individual lengths (this behaviour was also visible across all population sizes). This shows that GAUGE does not suffer a performance loss from its genotype to phenotype mapping. We can also see that the messyGA with the STD settings has a lower performance than both the simple GA and GAUGE, and its performance gets worse relative to those two systems with longer strings.

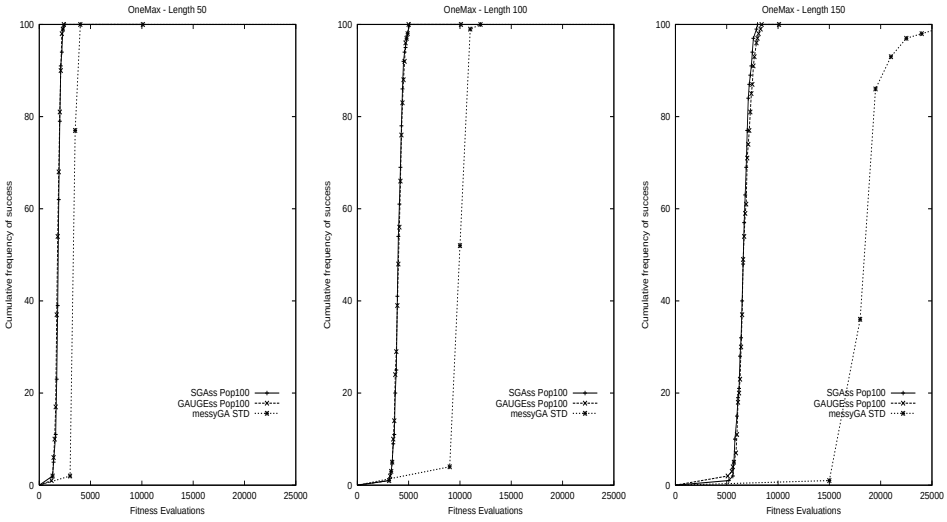


**Fig. 1.** Results for onemax problem.

### 4.2 Mastermind - a deceptive ordering version

In the original Mastermind game, one player is the *codebreaker* and tries to deduce a hidden configuration of four coloured pins, set by the *codemaker*, by having a maximum of ten guesses at it. The pins come in six different colours. Each guess has a score of black and white markers, a black marker indicating that one of the pins has the right colour and is in the right position, and a white marker indicating that a pin has the right colour, but is badly placed. More information can be found online at [10]. There have been some attemps at applying genetic algorithms to solve this game [1].

Some deceptive ordering problems have been defined in earlier work (check [7] and [8] for examples), and with similarities to Mastermind. We wanted, however, a problem with the following characteristics:

- easy to implement and quick to evaluate;
- easy to scale difficulty;
- defined search space with local optima;
- illustrates the ordering performance of algorithms.

Our version of Mastermind is therefore slightly different. First of all, we work with numbers to represent colours, and the information available to the *codebreaker* is somewhat reduced. When a combination of pins is to be evaluated, it receives one point for each pin that has the right colour, and if all pins are in the correct order then an additional fitness point is attributed - in other words, information about the correct placement of pins is only given if the whole string is correct. Some examples are shown in Table 3.

There is a deceptive aspect to this problem. If an individual is composed of all the correct pins but in the wrong order, it has reached a local optimum; since at least two pins are wrongly placed, then the global optimum is always at least at a hamming distance of two from every local optimum (since at least two values will need to be changed).

There is a good degree of control over the size and shape of the search space. By recalling combinatorial notions, we see that the size of the search space (i.e. all possible combinations of $p$ pins using $c$ colours) is given by $c^p$.

| Solution: 3 2 1 3 | |
|---|---|
| Individual | Fitness |
| 0 2 1 0 | 2 points |
| 0 1 2 0 | 2 points |
| 2 1 2 0 | 2 points |
| 3 1 2 0 | 3 points |
| 3 1 2 3 | 4 points |
| 3 2 1 3 | 5 points |

**Table 3.** Examples of evaluation of individuals using the Mastermind fitness function.

We can also calculate the number of local optima by using the formula:

$$\frac{p!}{\prod_{i=0}^{n}(x_i)!}$$

where
$n$ = number of colours in solution
$x_i$ = number of times colour $i$ appears in solution

**Results** We start by comparing GAUGE to an ordinary GA. To avoid redundant encoding in the simple GA, we used only cases of four and eight colours (encoded by two and three bits per codon, respectively). All the different GA settings (as described in Table 1) were again tested, but using the mentioned bits per codon to encode colours with the simple GA. We then used combinations with four, six, eight and ten pins for each number of colours. The solutions used for each combination are shown in Table 4, and were created using random strings of numbers from 0 to $c$, from which the first $p$ elements were extracted.

Our results showed GAs with generational replacement behaved better with small population sizes, whereas steady-state GAs worked better with larger ones. We also tested the statistical significance of the performance of GAUGE and the GAs using steady-state, on the average fitness and best fitness across all runs using a student t-test, and a bootstrap test for confirmation.

The results showed GAUGE outperforming all GA flavours across all the experiments, with a statistically significant difference on the majority of the experiments; table 5 shows the percentage of successful runs, after 25000 evaluations, for SGAss and GAUGE, with all population sizes, on the tested combinations of colours and pins. Example results for 4 colours, 8 pins, are shown in Fig. 2, with population sizes 100 and 800. These graphs plot the number of fitness function calls versus the cumulative frequency of success of 100 different runs.

We then moved onto comparing the messyGA with GAUGE, and GAUGE again outperformed the messyGA accross all its settings (as shown in Table 2). However, since the messyGA encodes positions for bits, whereas GAUGE encodes positions for values (which can directly represent colours), this gives GAUGE an

| | |
|---|---|
| 4 Colours, 4 pins | 3 2 1 3 |
| 4 Colours, 6 pins | 3 2 1 3 1 3 |
| 4 Colours, 8 pins | 3 2 1 3 1 3 2 0 |
| 4 Colours, 10 pins | 3 2 1 3 1 3 2 0 1 1 |
| 8 Colours, 4 pins | 7 6 1 3 |
| 8 Colours, 6 pins | 7 6 1 3 1 7 |
| 8 Colours, 8 pins | 7 6 1 3 1 7 2 4 |
| 8 Colours, 10 pins | 7 6 1 3 1 7 2 4 1 5 |

**Table 4.** Solutions used across all systems for each colours/pins combination.

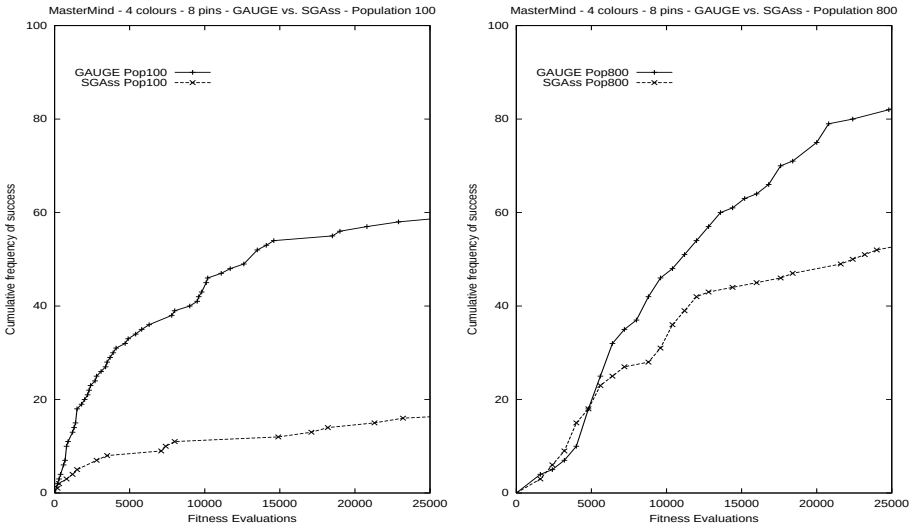


**Fig. 2.** Simple GA vs. GAUGE, MASTERMIND with 4 colours, 8 pins.

advantage. We therefore devised a new version of GAUGE (which we shall refer to as GAUGE1BIT) to encode positions for bits, rather than for full values, and compared both systems.

GAUGE1BIT showed a better performance than all the messyGA settings, as can be seen in Fig. 3, which shows the cumulative frequency of success of 100 runs plotted against the number of function calls. Table 6 shows the percentage of successful runs after 25000 evaluations, for all combinations of colours and pins tested. We noticed that allelic and genic mutation seemed to increase the performance of the messyGA, and setting OPT3, with thresholding and tiebreaking enabled, had a slightly better performance than all other messyGA settings.

| | | SGAss | | | | | | GAUGE | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Population size: | | 50 | 100 | 200 | 400 | 800 | 1600 | 50 | 100 | 200 | 400 | 800 | 1600 |
| 4colours | 4 pins | 96% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% |
| 4colours | 6 pins | 59% | 87% | 99% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% |
| 4colours | 8 pins | 11% | 16% | 26% | 32% | 52% | 63% | 45% | 58% | 81% | 83% | 82% | 79% |
| 4colours | 10 pins | 0% | 2% | 2% | 9% | 11% | 6% | 14% | 20% | 20% | 21% | 24% | 17% |
| 8colours | 4 pins | 43% | 64% | 83% | 97% | 99% | 100% | 95% | 100% | 100% | 100% | 100% | 100% |
| 8colours | 6 pins | 8% | 18% | 44% | 74% | 86% | 96% | 74% | 94% | 99% | 100% | 100% | 100% |
| 8colours | 8 pins | 0% | 2% | 0% | 2% | 2% | 6% | 7% | 15% | 16% | 14% | 10% | 6% |
| 8colours | 10 pins | 0% | 0% | 0% | 0% | 0% | 0% | 1% | 1% | 0% | 1% | 0% | 0% |

**Table 5.** Percentage of successful runs after 25000 evaluations, for SGAss and GAUGE, for all tested combinations of colours/pins, across all population sizes.

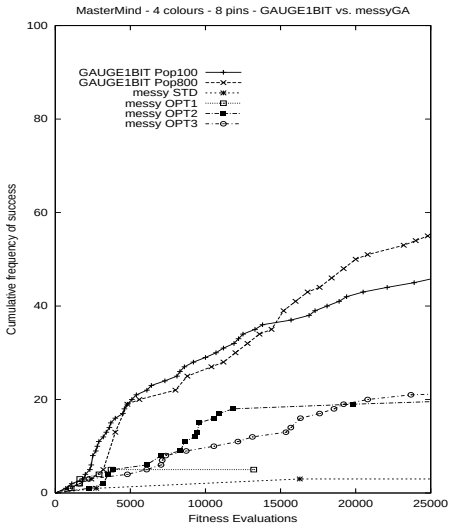


**Fig. 3.** GAUGE1BIT vs. messyGA, MASTERMIND with 4 colours, 8 pins.

## 5 Discussion

The results reported for the onemax problem show that GAUGE is a system that works. It performs no worse than a simple GA for that problem, showing that it doesn't suffer a performance loss from its genotype to phenotype mapping, or from its position-independence implementation. Moreover, no significant computational effort is required, due to the use of a simple three rule grammar.

Regarding the Mastermind results, they show an encouraging performance from GAUGE. Either encoding the position of values (directly transformed in colours) or encoding bits (later combined to form colours), GAUGE outperformed all other systems analysed, across a range of increasingly difficult com-

| | | messyGA | | | | GAUGE1BIT | |
|---|---|---|---|---|---|---|---|
| | Setting: | STD | OPT1 | OPT2 | OPT3 | Pop100 | Pop800 |
| 4colours | 4 pins | 73% | 72% | 94% | 97% | 100% | 100% |
| 4colours | 6 pins | 41% | 41% | 81% | 96% | 100% | 100% |
| 4colours | 8 pins | 3% | 5% | 19% | 21% | 45% | 55% |
| 4colours | 10 pins | 1% | 1% | 7% | 4% | 6% | 6% |
| 8colours | 4 pins | 27% | 27% | 50% | 84% | 88% | 100% |
| 8colours | 6 pins | 6% | 8% | 21% | 22% | 22% | 56% |
| 8colours | 8 pins | 1% | 0% | 0% | 0% | 0% | 0% |
| 8colours | 10 pins | 0% | 0% | 0% | 0% | 0% | 0% |

**Table 6.** Percentage of successful runs after 25000 evaluations, comparing all tested settings of messyGA with GAUGE1BIT, for all combinations of colours/pins.

binations of colours and pins, thus managing to escape the deception of local optima. Moreover, this was achieved without any parameter tuning, and the results were only limited by the number of fitness function calls. It would be interesting to increase that number, and see how far GAUGE can go.

## 6 Conclusions and Future Work

One of the known downfalls of genetic algorithms is the lack of position specification, which leads to convergence to local optima on ordering problems. By applying the principles behind Grammatical Evolution, we have established a new system, GAUGE. With its genotype/phenotype distinction, position/value separation, and small overhead processing required, GAUGE was successful on both a typical GA benchmark, Onemax, and a new deceptive problem introduced in this paper, Mastermind.

Further work will involve testing GAUGE on other problems, from harder versions of Mastermind (to test scalability) to linkage problems, and comparing GAUGE to other more recent and complex competent genetic algorithms.

## References


1. Bernier, J. L., Ilia Herraiz, C. Merelo, J. J., Olmeda, S., Prieto, A. 1996. Solving MasterMind using GAs and simulated annealing: a case of dynamic constraint optimization. In Proceedings PPSN, Parallel Problem Solving from Nature IV, LNCS 1141, pp. 554-563. Springer-Verlag.
2. Deb, K., Goldberg, D. E. 1991. mGA in C: A Messy Genetic Algorithm in C. Illinois Genetic Algorithms Laboratory (IlliGAL), report no. 91008.
3. Goldberg, David E. 1989. Genetic Algorithms in Search, Optimization and Machine Learning. Addison Wesley.
4. 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
5. Holland, J. H. 1975. Adaptation in Natural and Artificial Systems. Ann Arbor, MI: University of Michigan Press.
6. IlliGAL website. `http://www-illigal.ge.uiuc.edu/`.
7. Kargupta, H., Deb, K., Goldberg, D. E. 1992. Ordering genetic algorithms and deceptions. In Parallel Problem Solving from Nature - PPSN II (pp. 47-56).
8. Knjazew, D., and Goldberg, D. E. 2000. OMEGA - Ordering Messy GA : Solving Problems with the Fast Messy Genetic Algorithm and Random Keys. In GECCO-2000: Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufman, pp. 181-188
9. Koza, J. 1992. *Genetic Programming*. MIT Press.
10. Nelson, T. 1999. Investigations into the Master Mind Board Game. Website: `http://www.tnelson.demon.co.uk/mastermind/index.html`.
11. O'Neill M., Ryan C. 2001. Grammatical Evolution. IEEE Trans. Evolutionary Computation Vol. 5 No. 4, August 2001.
12. Ryan C., Collins J.J., O'Neill M. 1998. Grammatical Evolution: Evolving Programs for an Arbitrary Language. *LNCS 1391, Proceedings of the First European Workshop on Genetic Programming*, pages 83-95. Springer-Verlag.