A Grammatical Genetic Programming approach to modularity in Genetic Algorithms

Erik Hemberg[1], Conor Gilligan[1], Michael O'Neill[1] & Anthony Brabazon[2]

[1] UCD Natural Computing Research & Applications
School of Computer Science and Informatics
University College Dublin, Ireland
erik.hemberg@ucd.ie, conor.gilligan@ucd.ie, m.oneill@ucd.ie
[2] UCD Natural Computing Research & Applications
School of Business
University College Dublin, Ireland
anthony.brabazon@ucd.ie

Abstract. The ability of Genetic Programming to scale to problems of increasing difficulty operates on the premise that it is possible to capture regularities that exist in a problem environment by decomposition of the problem into a hierarchy of modules. As computer scientists and more generally as humans we tend to adopt a similar divide-and-conquer strategy in our problem solving. In this paper we consider the adoption of such a strategy for Genetic Algorithms. By adopting a modular representation in a Genetic Algorithm we can make efficiency gains that enable superior scaling characteristics to problems of increasing size. We present a comparison of two modular Genetic Algorithms, one of which is a Grammatical Genetic Programming algorithm, the meta-Grammar Genetic Algorithm (mGGA), which generates binary string sentences instead of traditional GP trees. A number of problems instances are tackled which extend the Checkerboard problem by introducing different kinds of regularity and noise. The results demonstrate some limitations of the modular GA (MGA) representation and how the mGGA can overcome these. The mGGA shows improved scaling when compared the MGA.

1 Introduction

In the natural world examples of modularity and hierarchies abound, ranging the biological evolution of cells to form tissues and organs to the physical structure of matter from the sub-atomic level up. In most examples of problem solving by humans, regularities in the problem environment are exploited in a divide-and-conquer approach through the construction of sub-solutions, which may then be reused and combined in a hierarchical fashion to solve the problem as a whole. Similarly Genetic Programming provides as components of its problem solving toolkit the ability to automatically create, modify and delete modules, which can be used in a hierarchical fashion. The objectives of this study are to investigate the adoption of principles from Genetic Programming [1] such as modularity and reuse (see Chapter 16 in [2]) for application to Genetic Algorithms, and to

couple these to an adaptive representation that allows the type and usage of these principles to be evolved through the use of evolvable grammars. The goal being the development of an evolutionary algorithm with good scaling characteristics, and an adaptable representation that will facilitate its application to noisy, dynamic, problem environments. To this end a grammar-based Genetic Programming approach is adopted, in which the grammars represent the construction of syntactically correct genotypes of the Genetic Algorithm. In particular, we compare the representations and performance of the meta-Grammar Genetic Algorithm (mGGA) [3] to the Modular Genetic Algorithm (MGA) [4], highlighting some of the MGA's representational limitations, and demonstrate the potential of a more expressive representation in the form of the mGGA to scale to problems of increasing size and difficulty. Additionally, we consider the introduction of noise into the Checkerboard problem, in order to assess how the representations might generalise into noisy, real-world problem domains. The remainder of the paper is structured as follows. Section 2 provides background on earlier work in modular GAs and describes the meta-Grammar Genetic Algorithm. Section 3 details the experimental approach adopted and results, and finally section 4 details conclusions and future work.

2 Background

There has been a large body of research on modularity in Genetic Programming and effects on its scalability, however the same cannot be stated for the Genetic Algorithm (GA). In this section we present two modular representations as implemented in the Modular GA [4] and the meta-Grammar GA [3].

2.1 Modular Genetic Algorithm

Garibay et al. introduced the Modular Genetic Algorithm, which was shown to signficantly outperform a standard Genetic Algorithm on a scalable problem with regularities [4]. The genome of an MGA individual is a vector of genes, where each gene is comprised of two components, the `number-of-repetitions` and some `function` which is repeated according to the value of the repetitions field. For example, if we had a function (`one()`) that always returned the value `1` when called and another (`zero()`) that returned the value `0` we have a representation that can generate binary strings. A sample individual comprised of three genes might look like: `{2, zero()}, {4, one()}, {2, zero()}`, which would produce the binary string `00111100`. The MGA was shown to have superior ability to scale to problems of increasing complexity than a standard GA.

2.2 Grammatical Evolution by Grammatical Evolution

The grammar-based Genetic Programming approach upon which this study is based is the Grammatical Evolution by Grammatical Evolution algorithm [5], which is in turn based on the Grammatical Evolution algorithm [6–9]. This is

a meta-Grammar Evolutionary Algorithm in which the input grammar is used to specify the construction of another syntactically correct grammar. The generated grammar is then used in a mapping process to construct a solution. In order to allow evolution of a grammar (Grammatical Evolution by Grammatical Evolution $(GE)^2$), we must provide a grammar to specify the form a grammar can take. This is an example of the richness of the expressiveness of grammars that makes the GE approach so powerful. See [6, 10, 11] for further examples of what can be represented with grammars and [12] for an alternative approach to grammar evolution. By allowing an Evolutionary Algorithm to adapt its representation (in this case through the evolution of the grammar) it provides the population with enhanced robustness in the face of a dynamic environment, in particular, and also to automatically incorporate biases into the search process. In this case we can allow the meta-Grammar Genetic Algorithm to evolve biases towards different building blocks of varying sizes. In this approach we therefore have two distinct grammars, the *universal grammar* (or grammars' grammar) and the *solution grammar*. The notion of a universal grammar is adopted from linguistics and refers to a universal set of syntactic rules that hold for spoken languages [13]. It has been proposed that during a child's development the universal grammar undergoes modifications through learning that allows the development of communication in their parents native language(s) [14]. In $(GE)^2$ the universal grammar dictates the construction of the solution grammar. In this study two separate, variable-length, genotypic binary chromosomes were used, the first chromosome to generate the solution grammar from the universal grammar and the second chromosome generates the solution itself. Crossover operates between homologous chromosomes, that is, the solution grammar chromosome from the first parent recombines with the solution grammar chromosome from the second parent, with the same occurring for the solution chromosomes. In order for evolution to be successful it must co-evolve both the meta-Grammar and the structure of solutions based on the evolved meta-Grammar, and as such the search space is larger than in standard Grammatical Evolution.

2.3 meta-Grammars for Bitstrings

A simple grammar for a fixed-length (8 bits in the following example) binary string individual of a Genetic Algorithm is provided below. In the generative grammar each bit position (denoted as `<bit>`) can become either of the boolean values. A standard variable-length Grammatical Evolution individual can then be allowed to specify what each bit value will be by selecting the appropriate `<bit>` production rule for each position in the `<bitstring>`.

```
<bitstring> ::= <bit><bit><bit><bit><bit><bit><bit><bit>
<bit> ::= 1 | 0
```

The above grammar can be extended to incorporate the reuse of groups of bits (building blocks). In this example all building blocks that are mutliples of two are provided, although it would be possible to create a grammar that adopted more complex arrangements of building blocks.

```
<bitstring> ::= <bbk4><bbk4> | <bbk2><bbk2><bbk2><bbk2>
              | <bbk1><bbk1><bbk1t><bbk1><bbk1><bbk1><bbk1><bbk1>
<bbk4> ::= <bit><bit><bit><bit>
<bbk2> ::= <bit><bit>
<bbk1> ::= <bit>
<bit> ::= 1 | 0
```

The above grammars are static, and as such can only allow one building block of size four and of size two in the second example. It would be better to allow our search algorithm the potential to uncover a number of building blocks of any one size from which a Grammatical Evolution individual could choose from. This would facilitate the application of such a Grammatical GA to:

– problems with more than one building block type for each building block size,
– to search on one building block while maintaining a *reasonable* temporary building block solution,
– and to be able to switch between building blocks in the case of dynamic environments.

All of this can be achieved through the adoption of meta-Grammars as were adopted earlier in [5]. An example of such a grammar for an 8-bit individual is given below.

```
<g> ::=  "<bitstring> ::=" <reps>
              "<bbk4> ::=" <bbk4t>
              "<bbk2> ::=" <bbk2t>
              "<bbk1> ::=" <bbk1t>
               "<bit> ::=" <val>

<bbk4t> ::= <bit><bit><bit><bit>
<bbk2t> ::= <bit><bit>
<bbk1t> ::= <bit>
<reps> ::= <rept> | <rept> "|" <reps>
<rept> ::= "<bbk4><bbk4>" | "<bbk2><bbk2><bbk2><bbk2>"
         | "<bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1>"
<bit> ::= "<bit>" | 1 | 0
<val> ::= <valt>  | <valt> "|" <val>
<valt> ::= 1 | 0
```

In this case the grammar specifies the construction of another generative bitstring grammar. The subsequent bitstring grammar that can be produced from the above meta-grammar is restricted such that it can contain building blocks of size 8. Some of the bits of the building blocks can be fully specified as a boolean value or may be left as unfilled for the second step in the mapping process. An example bitstring grammar produced from the above meta-grammar could be:

```
<bitstring> ::= <bit>11<bit>00<bit><bit> | <bbk2><bbk2><bbk2><bbk2>
              | 11011101 | <bbk4><bbk4> | <bbk4><bbk4>
<bbk4> ::= <bit>11<bit>
<bbk2> ::= 11
<bbk1> ::= 1
<bit> ::= 1 | 0 | 0 | 1
```

To allow the creation of multiple building blocks of different sizes the following grammar could be adopted (again shown for 8-bit strings).

```
<g> ::=  "<bitstring> ::=" <reps>
              "<bbk4> ::=" <bbk4>
              "<bbk2> ::=" <bbk2>
              "<bbk1> ::=" <bbk1>
               "<bit> ::=" <val>

<bbk4> ::= <bbk4t> | <bbk4t> "|" <bbk4>
<bbk2> ::= <bbk2t> | <bbk2t> "|" <bbk2>
<bbk1> ::= <bbk1t> | <bbk1t> "|" <bbk1>
<bbk4t> ::= <bit><bit><bit><bit>
<bbk2t> ::= <bit><bit>
<bbk1t> ::= <bit>
<reps> ::= <rept> | <rept>  "|" <reps>
<rept> ::= "<bbk4><bbk4>" | "<bbk2><bbk2><bbk2><bbk2>"
         | "<bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1>"
<bit> ::= "<bit>" | 1 | 0
<val> ::= <valt> | <valt> "|" <val>
<valt> ::= 1 | 0
```

An example bitstring grammar produced by the above meta-grammar is provided below.

```
<bitstring> ::= <bit>11<bit>00<bit><bit> | <bbk2><bbk2><bbk2><bbk2>
              | 11011101 | <bbk4><bbk4> | <bbk4><bbk4>
<bbk4> ::= <bit>11<bit> | 000<bit>
<bbk2> ::= 11 | 00 | <bit>1
<bbk1> ::= 0 | 0
<bit> ::= 1 | 0 | 0 | 1
```

Modularity exists above in the ability to specify the size and content (or partial content) of a buiding block through its incorporation into the solution grammar. This building block can then be repeatedly reused in the generation of the phenotype. An additional mechanism for reuse is through the Wrapping operator of Grammatical Evolution. During the mapping process if we reach the end of the genotype and still have outstanding decisions to make on the construction of our phenotype we can invoke the wrapping operator to move our reading head back to the first codon in the genome. This allows the reuse of rule choices if the codons are used in the same context. Given that the lengths of binary strings which may need to be represented can grow quite large it is possible to automate the creation of meta-grammars by simply providing the length of the target solution and creating all possible building block structures that can be used to create a bitstring of the target length. In this study the target binary strings are of lengths 60, 90, 120, 180, and 210. The building block sizes incorporated in their corresponding grammars are therefore all integers that divide into the target string lengths (i.e., for a target string of length 60 the building blocks are of sizes 30, 20, 15, 12, 10, 6, 5, 4, 3, 2 and 1). Meta-grammars are of course not limited to the specification of grammars for binary strings and can be easily extended to the representation of real and integer strings as well as programs, or any structure which can be represented in a grammatical form.

3 Experimental Setup and Results

Before detailing the experimental design and setup we first introduce the problems on which we will benchmark the two representations under investigation.

3.1 The Checkerboard-Pattern Discovery Problem

Given the lack of suitable benchmark problems in the Genetic Algorithm literature that consider modularity, Garibay et al., [4] proposed the Checkerboard-Pattern discovery problem. In this problem a pattern of colours or states is imposed upon a two dimensional grid called the Checkerboard. There are 2 possible states adopted for each square on the grid, i.e., black or white, which can be represented as bit values 1 and 0 respectively. Each candidate solution tries to recapture the pattern contained in the target Checkerboard. Fitness is simply measured by summing the number of squares that contain the correct state. In this study we then normalise fitness to the range 0.0 to 1.0, and standardise fitness such that 0.0 is the best possible fitness where all of the candidate solution's squares exactly match the target checkerboard-pattern. It is easily possible to scale the problem in terms of its complexity, modularity and regularity by increasing the size of the checkerboard, the number of patterns, and changing the number of components in each pattern, respectively. Example instances of this problem which are adopted in this study and in [4] are presented in Fig.1, which illustrates scaled-up versions of a 4X8 pattern to 8X16 and 16X32. Another problem instance tackled in this study of a 8X16 checkerboard pattern is also illustrated. A third set of problem instances are examined which add noise to the state of each sqaure upon the evaluation of each individual. This is implemented by randomly switching the state of a square with a predefined probability for the patterns already presented in Fig.1. With the addition of noise to the regular patterns this makes it more challenging to uncover the underlying patterns and thus add an additional element of real-world interest to this benchmark problem. The amount of noise can easily be tuned by altering the probability of error.

3.2 Comparing Performance of mGGA and MGA

Table 1. Performance changes for the mGGA on the standard non-noisy problem instances. The average best fitness after 500 generations is 0.019792 for 2^{4X8} and after a 1000 generations 0.019531 for 2^{8X16}. The differnece in fitness between the two generations is 0.000261. The average best fitness after 400 generations for 2^{16X32} is 0.01875. Difference between 2^{4X8} and 2^{16X32} is 0.001042.

	Performance drop (% of fitness decrease)	
Complexity increase	MGA	mGGA
from 2^{4X8} to 2^{8X16}	3.68%	0.02%
from 2^{4X8} to 2^{16X32}	11.38%	0.1%

30 runs on each problem instance were performed with the mGGA using a population size of 1000, tournament selection (size 3), mutation probability of 0.001 per gene, and crossover probability of 0.7. The number of generations was selected to reflect the values adopted in Garibay et al's study [4], i.e. 500 for the

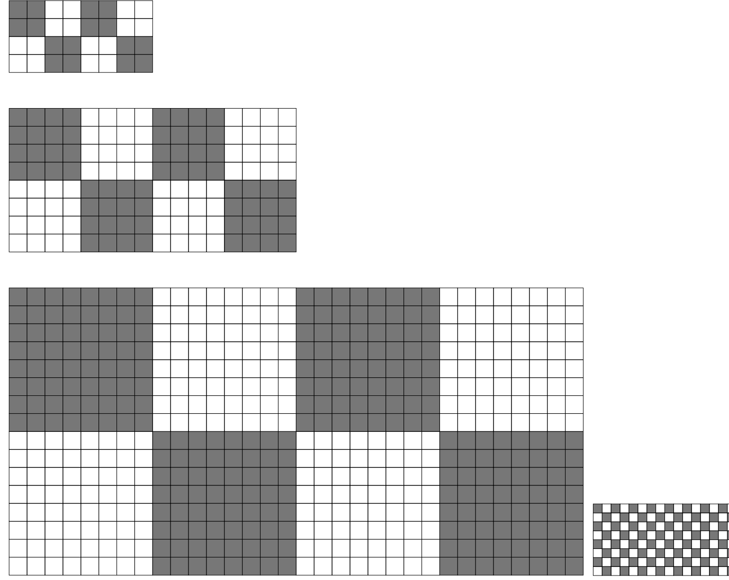

Fig. 1. The original checkerboard-pattern matching problem instances (from left 2^{4X8}, 2^{8X16} and 2^{16X32}) as presented in [4]. On the far right is a new 2^{8X16} checkerboard-pattern matching problem checkerboard instance with finer-grained regularity.

Table 2. Performance values for the mGGA on the standard non-noisy problem instances. Average values are for 30 runs. The value in parenthsis is random search for 1000000 tries.

Problem	Best fitness	Mean fitness	Variance(best fit.)	Successful Runs
2^{4X8}	0.0119	0.0168	0.0017	26/30
2^{8X16}	0.0211 (0.164)	0.0265 (0.25)	0.0030	25/30
2^{16X32}	0.0188 (0.416)	0.0034 (0.5)	0.0248	27/30

4X8, 1000 for the 8X16 and 2000 generations for the 16X32 problem instance. Random initialisation was used, and the fitness values and their variance reported in the initial population averaged over the 30 runs reflects this (see e.g. Fig. 2). The results are reported in Fig 2, with the percentage gains in performance and fitness statistics reported in Tables 1 and 2 respectively. It is clear that as the problem instances increase in complexity there are economies of scale to be achieved, with the relative performance of the mGGA improving significantly with each jump in problem size. An additional problem instance as portrayed in Fig.1(far right) was examined with the same parameters, with the results presented in Fig.3. In this case the pattern is of size 1X1 and as such is much finer grained than the patterns examined earlier. It is difficult for the MGA to efficiently represent a solution to this problem instance due to the nature of the pattern. Effectively each squares state must be specified individualy. However,

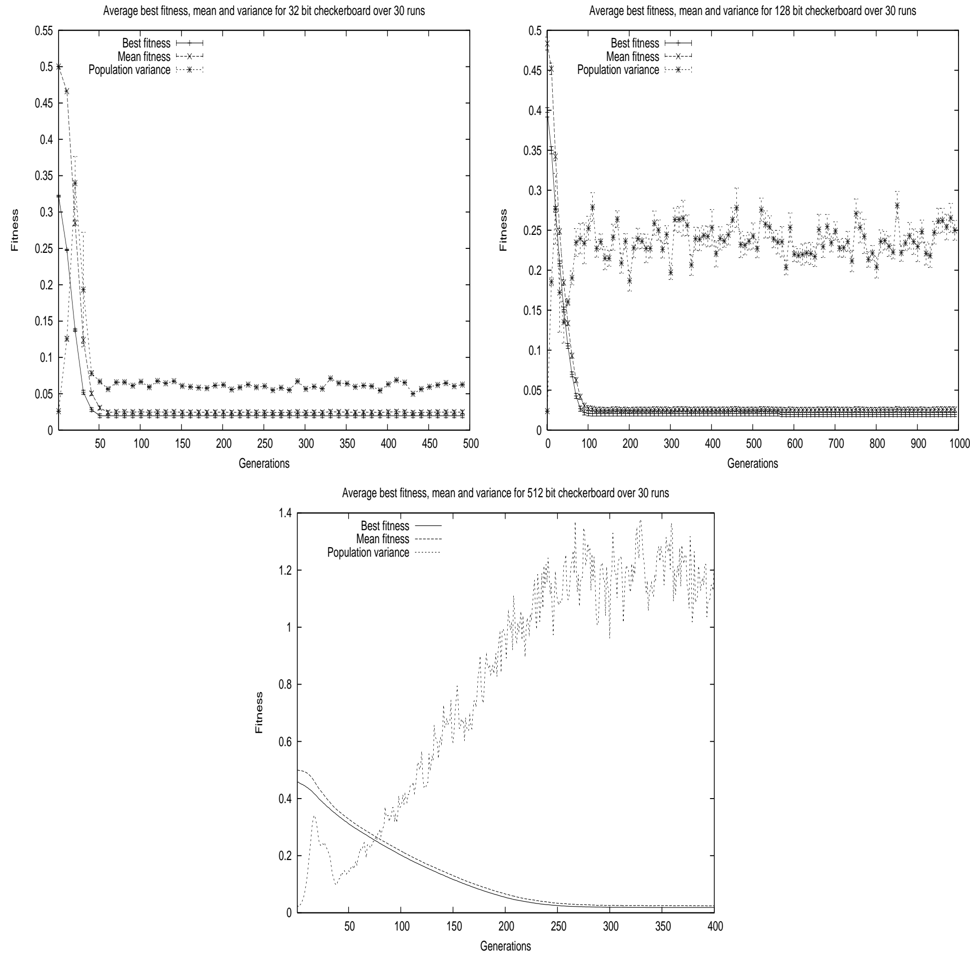

Fig. 2. A graph for the mGGA on 2^{4X8} (top left), 2^{8X16} (top right), and 2^{8X32} instances (bottom).

this is not the case with the mGGA which can encode effectively parameterise the evolved modules to specify multiple square states with different values.

3.3 mGGA Performance Under Noisy Conditions

In order to gain some preliminary insight into the performance of the mGGA in a more realistic real-world setting it was decided to conduct experimental runs incorporating noise into the target patterns. This was achieved by simply flipping each bit in the target pattern with probability p_n. Runs were conducted using the same parameters as previously described for noise probabilities $p_n = 0.05, 0.075$ on the 2^{128} sized problem. The results achieved here are presented

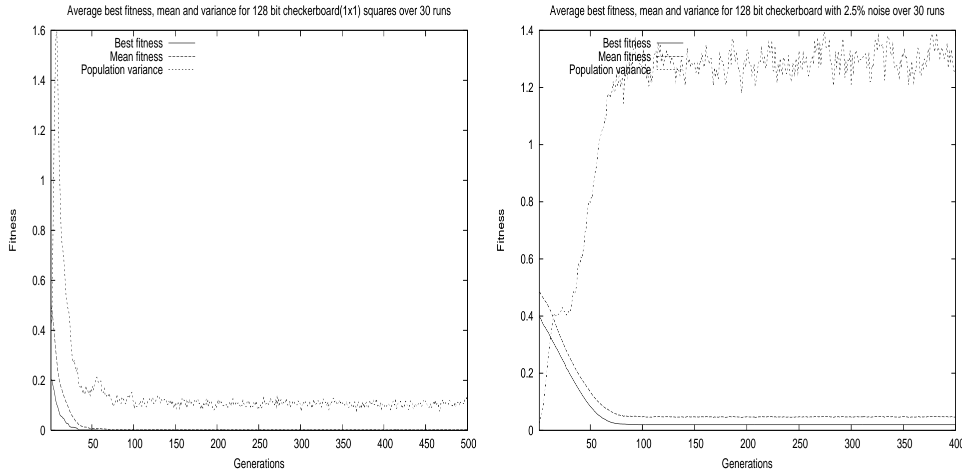

Fig. 3. A graph for the mGGA on 2^{8X16} checkerboard (1X1) (left) and, on the right the standard 2^{8X16} instance with 2.5% noise.

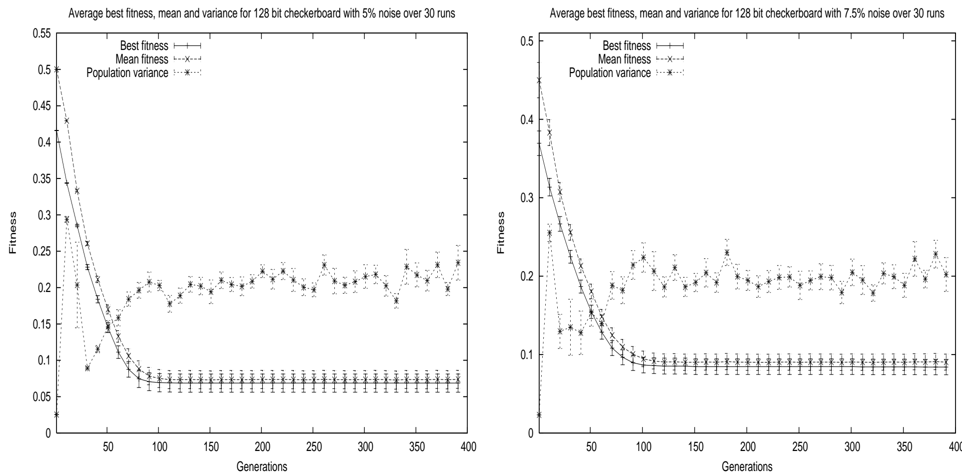

Fig. 4. A graph for the mGGA on 2^{8X16} with 5% noise (left) and the 2^{8X16} with 7.5% noise (right).

in Table 3 and Fig.4. As can be expected, the addition of noise reduced the algorithm performance on average, however on inspection of individual runs it was seen that this performance drop was manifest in an increased, but still small, number of runs which failed to converge to an optimal solution; instead converging prematurely on areas of very poor fitness. This indicates that the population may be converging too quickly in the early stages of the algorithm, loosing whatever diversity was present in the initial population. It is possible

Table 3. Statistics for performance of the mGGA on the Noisy Checkerboard-Pattern Discovery instances.

Noise level	Best fitness	Mean fitness	Variance(best fit.)	Successful Runs
$S(1x1)$ $p = 0$	0	0.0024	0	30/30
$p = 0$	0.0195	0.0253	0.0027	25/30
$p = 0.025$	0.0198	0.0468	0.002	24/30
$p = 0.05$	0.0664	0.0719	0.0123	22/30
$p = 0.075$	0.0841	0.0904	0.0098	13/30

that through adjusting parameters of the EA better results could be achieved. It may also be wise to examine the initialization technique as it is possible that the initial population lacks diversity. The increase in search space size with the use of both meta-grammar and solution chromosomes may also be having an impact on performance.

4 Conclusions & Future Work

We presented a comparison of the meta-Grammar GA (mGGA) to the Modular GA (MGA), illustrating the application of evolvable grammars to implement modularity in Genetic Algorithms. We also introduced a number of variations to the benchmark Checkerboard-Pattern discovery problem including different types of regularity and the introduction of noise to bring the benchmark closer to real-world scenarios. On the problem instances examined there are clear performance advantages for the mGGA when compared to the MGA. In addition to the application to more benchmark problem instances in particular to those belonging to the dynamic class, future work will investigate the effects alternative grammars and comparisons to other GAs from the literature including the competent GAs. A number of avenues to facilitate the co-evolution of the grammar and solution, such as different operator probabilities, will also be investigated.

5 Acknowledgement

This research is based upon works supported by the Science Foundation Ireland under Grant No. 06/RFP/CMS042.

References

1. Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA.
2. Koza, J. R., Keane, M. A., Streeter, M. J., Mydlowec, W., Yu, J., Lanza, G. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence.* Kluwer Academic Publishers.

3. O'Neill M., Brabazon A. (2005). mGGA: The Meta-Grammar Genetic Algorithm. *LNCS 3447 Proceedings of the European Conference on Genetic Programming EuroGP 2005*, pp.311-320. Springer.
4. Garibay O.O., Garibay I.I., Wu A.S. (2003). The Modular Genetic Algorithm: Exploiting Regularities in the Problem Space. *In LNCS 2869 Computer and Information Science ISCIS 2003*, pp.584-591. Springer.
5. O'Neill, M., Ryan, C. (2004). Grammatical Evolution by Grammatical Evolution. The Evolution of Grammar and Genetic Code. *LNCS 3003. Proc. of the European Conference on Genetic Programming 2004*, pp. 138-149. Springer.
6. O'Neill, M., Ryan, C. (2003). *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers.
7. O'Neill, M. (2001). Automatic Programming in an Arbitrary Language: Evolving Programs in Grammatical Evolution. PhD thesis, University of Limerick.
8. O'Neill, M., Ryan, C. (2001). Grammatical Evolution, *IEEE Trans. Evolutionary Computation*. 2001.
9. Ryan, C., Collins, J.J., O'Neill, M. (1998). Grammatical Evolution: Evolving Programs for an Arbitrary Language.*Proc. of the First European Workshop on GP*, pp. 83-95, Springer-Verlag.
10. Dempsey, I., O'Neill, M., Brabazon, A. (2004). Grammatical Constant Creation. In LNCS 3103, *Proceedings of GECCO 2004*, Part 1, pp. 447-458, Seattle, USA.
11. O'Neill, M., Cleary, R. (2004). Solving Knapsack Problems with Attribute Grammars. In *Proceedings of the Grammatical Evolution Workshop 2004*, GECCO 2004, Seattle, USA.
12. Shan, Y., McKay, R. I., Baxter, R., Abbas, H., Essam, D., Nguyen, H.X. (2004). Grammar Model-based Program Evolution. In *Proceedings of the 2004 Congress on Evolutionary Computation, CEC 2004*, Vol. 1, pp. 478-485, Portland, USA.
13. Chomsky, N. (1975). *Reflections on Language*. Pantheon Books. New York.
14. Pinker, S. (1995). *The language instinct: the new science of language and the mind*. Penguin, 1995.