A Non-Destructive Grammar Modification Approach to Modularity in Grammatical Evolution

John Mark Swafford, Erik Hemberg, Michael O'Neill, and Miguel Nicolau

Natural Computing Research & Applications Group
Complex and Adaptive Systems Laboratory
School of Computer Science & Informatics
University College Dublin
Ireland

Anthony Brabazon

Natural Computing Research & Applications Group
Complex and Adaptive Systems Laboratory
School of Business
University College Dublin
Ireland

Abstract

Modularity has proven to be an important aspect of evolutionary computation. This work is concerned with discovering and using modules in one form of grammar-based genetic programming, grammatical evolution (GE). Previous work has shown that simply adding modules to GE's grammar has the potential to disrupt fit individuals developed by evolution up to that point. This paper presents a solution to prevent the disturbance in fitness that can come with modifying GE's grammar with previously discovered modules. The results show an increase in performance from a previously examined grammar modification approach and also an increase in performance when compared to standard GE.

1 Introduction

Modularity is an important open issue in the field of genetic programming [18], and has been studied in a variety of contexts. These range from examining abstract principles taken from biology [21] to the empirical analysis of the performance of different approaches to enabling and exploiting modularity in evolutionary computation. This research is classified under the latter. As modularity has been shown to be extremely useful for the scalability of evolutionary algorithms (Koza shows this for genetic programming [12]), it is important to understand the effects of different methods of encapsulating and exploiting modularity in these stochastic search methods.

Genetic algorithms (GAs) [9] and genetic programming (GP) [11] have been studied fairly extensively in this context. However, modularity in grammatical evolution (GE) [3, 17] has not been examined as thoroughly, and is the focus of this work. Studying modularity in the context of GE is especially interesting because of its genotype-to-phenotype mapping. The context-free grammar used in this process provides an easy method for reusing encapsulated information from GE's individuals. By examining the derivation trees created by this mapping process, "good" information discovered by GE can be identified, encapsulated, and placed directly into the grammar to be used in the appropriate context. Previous research by Swafford et al. [22] has shown the potential advantages and drawbacks of this approach to modularity in GE. Here, an improved method for enhancing GE's grammar with modules is proposed in

order to alleviate such drawbacks. To accomplish this, two methods of modifying the grammar will be examined:

1. modifying the grammar and remapping each individual with it's original genotype and the new grammar, and
2. modifying the grammar and each individual's genotype so the productions picked during the mapping process is the same as it was before the grammar modification.

The differences in their performance will be noted and compared to each other, as well as compared to standard GE's.

In this paper, a module is the sub-derivation tree of an individual which is considered to contain beneficial information (Refer to Figures 1(b) and 1(c) in Section 4 for examples of a derivation tree and a module). Once modules are discovered and encapsulated, they are inserted into a global grammar and may not be modified or evolved, unlike Koza's automatically defined functions(ADFs) [12]. Koza's ADFs are also parameterized and local to each individual, where the modules in this study are stored in a global grammar for all individuals.

The rest of the paper is structured as follows. The following section outlines some previous work relating to modularity in GP and GE. Section 3 describes the method used to identify modules. Next, Section 4 explains how the identified modules are incorporated into GE's grammar. Then, Section 5 presents the experimental setup, and Section 6 details the results of this work, as well as their meaning. Finally, Section 7 gives the conclusions and avenues for future work.

2 Previous Work

There have been many previous explorations into different methods of discovering, creating, and using modules in GP. Some of the earliest work in this area is that of Angeline and Pollack [1, 2]. They developed methods for picking out modules of useful information to be passed from individual to individual during an evolutionary run. They use *compress*, *expand*, and *atomization* operations to "lock" potentially useful parts of GP syntax trees, to "unlock" previously compressed portions of syntax trees, and to compress more information into previously compressed syntax trees. They show the first step in basic module encapsulation and how advantageous this can be, and how beneficial capturing these modules is during the course of an evolutionary run.

Next, the most popular and most studied of approaches to exploiting modularity in GP is Koza's ADFs [12]. ADFs are functions which are generated as separate branches of a GP individual's syntax tree. They are then used by that individual's result-producing branch (RPB) of the syntax tree. The most notable feature of ADFs is that they can be modified by evolutionary operators and parameterized to accept any number of parameters. This particular approach to modularity in GP has been shown to outperform standard GP on many benchmark problems. It has also been shown that ADFs are not beneficial on too simple of problems, and to see any significant increase in performance, the problem must be sufficiently difficult.

Keijzer et al. [10] also explore the notion of modularity, but on a different scale than most other work. Where most approaches to modularity focus on improving performance on a per-run basis, Keijzer and his colleagues use entire evolutionary trials to discover modules. They introduce run-transferable libraries (RTLs), which are lists of modules discovered over a number of independent runs and are used to seed the population of a new run. These RTLs show an increase in performance over standard GP, and greatly enhance the scalability of GP by training the RTLs on simple problems before using them for harder problems [20].

Further research on modularity in GP was carried out by Krawiec and Weiloch [13]. They define a new way to exploit modularity, called *functional modularity*. They attempt to discover modules without using the context of the problem as a whole. They use semantics to evaluate how good a module is without using that module in the original problem. They give the example of a battery in a flashlight. There may be a way to determine the quality of the battery without using it in the flashlight. The use of semantics is a very difficult problem and Krawiec and Weiloch state that further studies are needed to fully understand how to best exploit the semantics of modules.

Modularity has also been studied in the context of grammar-based forms of GP. Whigham [24] uses a non-mapping form of grammar-based GP to exploit modularity. He examines the parse trees created

by individuals and extracts sub-trees to enhance the grammar during the evolutionary run. He uses sub-trees from the fittest individuals and turns them into additional productions and/or rules to add to the grammar. His results showed that altering the grammar in this way led to an increase in the number of successful runs on the six-multiplexer problem.

Hemberg et al. [7] also study modularity in a grammar-based form of GP (grammatical evolution) which uses a mapping process to create individuals by implementing meta-grammars, or grammars which generate grammars, which are then used to solve the given problem, called GE^2. GE^2 is shown to have increased performance and scale better in comparison to the Modular Genetic Algorithm (MGA) [4] on problems known to have regularities (one example of such a problem is the checkerboard problem).

Approaches to enabling ADFs in grammar-based forms of GP have also been examined. Hemberg et al. [8] and O'Neill and Ryan [16] use ADFs in GE and achieve increases in performance over standard GE on certain problems (Santa Fe Ant Trail, Los Altos Ant Trail, and San Mateo Ant Trail), and decreases in performance on others. Similarly, Harper and Blair [5] use Dynamically Defined Functions (DDFs) with GE on the Minesweeper problem, showing that DDFs outperform standard GE, and GE with ADFs.

For a more in-depth review of previous work of modularity in GP, refer to the work by Walker and Miller [23] and Hemberg [6].

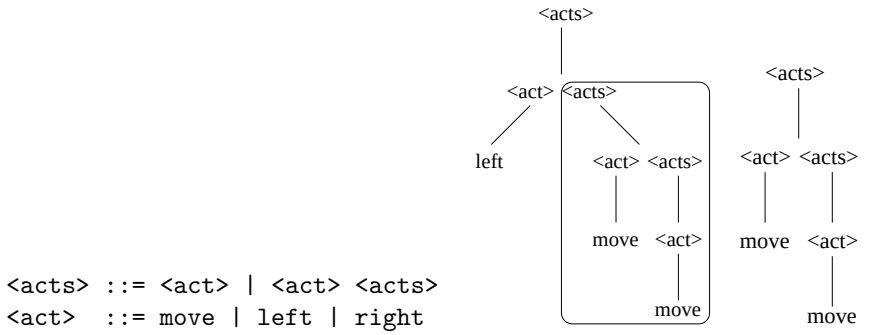
3 Module Identification

In this paper, the module identification approach is based on that adopted by Swafford et al. [22]. The first step in identifying a module is choosing a parent individual to provide a candidate module. This is done by iterating the population and allowing each individual to contribute one candidate module. Next, the candidate module must be evaluated. A node on the parent's derivation tree is randomly picked. The sub-derivation tree starting at this node is the candidate module, and the original fitness of the individual, f_0, is recorded. Next, n new derivation trees starting with the same root as the candidate module are randomly created and take turns replacing the candidate module in the parent. After each replacement, the fitness of the new parent individual is calculated, $f_{1...n}$. If f_0 is a better fitness than $p\%$ of $f_{1...n}$, the candidate module is saved for later use. When this is the case, the difference between f_0 and each of $f_{1...n}$ is taken and the average of these is the module's fitness value. This approach to module identification was inspired by that taken by Majeed and Ryan [14]. To keep the number of modules being incorporated into the grammar at a reasonable size, only the m (where $m > 0$) best modules are kept after each module identification step before they are used to modify the grammar. Some variations of the parameters n, p, and, m were tested in preliminary experiments, but there were no significant differences in performance between the variations examined.

4 Grammar Enhancement by Modules

Once modules have been identified, they need to be incorporated into the evolving population. To accomplish this, they are added to GE's grammar. Consider the simple grammar in Figure 1(a), an individual producible by that grammar (Figure 1(b)), and a module selected from that individual (Figure 1(c)). This module is incorporated into the grammar by taking the phenotype produced by this particular module (`move move`) and making it a production of the rule matching the module's root symbol. Since the module's root symbol is `<acts>`, a module library non-terminal is added to the rule matching the module's root node, and a new rule is created for this library non-terminal. Actual module non-terminals are added to the library non-terminal. A grammar modified in this way can be seen in Figure 1(d).

Swafford et al. [22] show how this approach to modifying the grammar in GE has the potential to be extremely destructive to the fitness of the entire population. In an attempt to remedy this, a "genotype repair" was also implemented. When the grammar is changed, it is highly likely that any given codon in an individual's genotype will not pick the same production as it did pre-grammar modification. This repair mechanism ensures the genotype maps to the same phenotype it did before the grammar was changed. Once the grammar has been modified and modules are being used in the population, it is possible, and actually very likely, that new modules will be discovered and will replace some or all of the previously discovered modules. When using the genotype repair method, this can cause a potential problem if an individual previously mapped to a module which no longer exists in the grammar. To



(a) A sample grammar (b) A sample individual (c) A sample module

```
<acts>         ::= <act> | <act> <acts>
                 | <acts_mod_lib>
<act>          ::= move | left | right
<acts_mod_lib> ::= <mod_0>
<mod_0>        ::= move move
```

(d) Grammar with modules

Figure 1: These figures show how a grammar is modified when a module is added to it.

remedy this problem, any module which appears in the population is allowed to remain in the grammar, even if it is not one of the twenty best modules kept at each module identification step. When this module no longer appears in the population it is then removed from the grammar.

5 Experimental Setup

The purpose of this work is to further examine grammar modification using modules. The experimental setup shown here compares three approaches:

1. **Remap:** Modules are added to the grammar and the population is remapped with the new grammar. When individuals are remapped, their genotypes remain unchanged, but the mapping process may (but is not required to) produce a new phenotype because of the altered grammar.

2. **Repair:** Modules are added to the grammar and the genotype of each individual is repaired so the productions picked during the mapping process with the new grammar are the same as the productions picked before the grammar was changed. It is possible that some individuals use modules that have been removed from the grammar. When this is the case, those modules are added back to the grammar and allowed to remain until individuals no longer use them.

3. The last variant is standard GE where the grammar is never changed.

These three variations of GE are used on five different benchmark problems: Santa Fe Ant Trail [8, 12], $x^5 - 2x^3 + x$ Symbolic Regression [19], 8×8 Lawn Mower, 12×12 Lawn Mower, and 14×14 Lawn Mower [12]. Koza [12], Walker and Miller [23], and Hemberg [8] all used different approaches to modularity to achieve improvements in performance on the Santa Fe Ant Trail. Both Koza [12] and Walker and Miller [23] demonstrated better performance on different symbolic regression problems and various Lawn Mower problems. The experimental parameters for these problems are shown in Table 1. The sub-derivation tree operators mentioned in Table 1 operate on GE individuals' derivation trees as opposed to the typical single-point crossover and int-flip mutation, which operate on GE individuals' genotypes. It should be noted that the implementation of GE used for this work is GEVA [15]. GEVA minimizes fitness, so in the following figures and tables, lower fitness values are always better.

Table 1: Experimental setup for all evolutionary runs unless otherwise noted

Parameter	**Value**
Generations	100
Population	500
Selection	Tournament (Size 5)
Wrapping	None
Crossover	Sub-derivation tree (80%)
Mutation	Sub-derivation tree (20%)
Elites	5
Initialization	Ramped Half and Half
Replacement	Generational
Max. Derivation Tree Depth	25 (100 for Lawn Mower problems)
Initial Depth	10
Trials	50

6 Results and Discussion

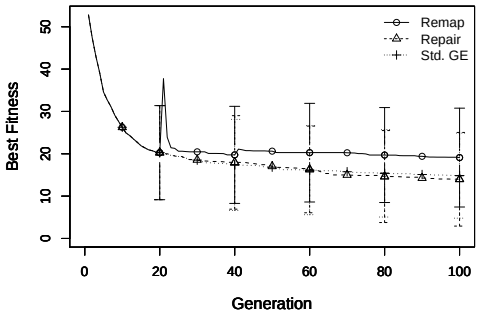
Modifying the grammar during an evolutionary run can potentially be highly destructive to the population's fitness, even if beneficial information is used for the modification [22]. This is due to the genotype-to-phenotype mapping process GE employs. By still modifying the grammar, but ensuring individuals keep their original mapping, good information can be encapsulated and put into GE's grammar, without destroying the information the population has already discovered. The results reported in this section demonstrate how the approaches outlined in Section 5 perform and compare to one another.

6.1 Grammar Modification and Fitness

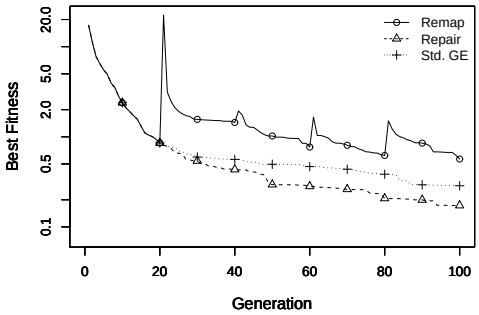
The primary goal of this research is to identify methods of encapsulating modules and using them in GE's grammar with no undesirable side effects and maximal benefits. To compare the methods presented here, the first characteristic to examine is how they impact the best fitness of a population during an evolutionary run. Figures 2(a) – 2(e) show the average best fitness achieved for each approach over the course of 50 evolutionary runs. The *Remap* lines represent GE using the grammar modification approach **without** genotype repair. The *Repair* lines represent GE using grammar modification and genotype repair. The *Std. GE* simply represent the standard GE setup. The errors bar on each graph are plotted using a 95% confidence interval, except Figure 2(b), which has no error bars due to the y-axis being a log scale.

One of the most obvious characteristics of Figure 2(a) and 2(b) is the occasional degradation in fitness every 20 generations on the *Remap* approach. The generations at which these degradations of fitness occur coincide with the generations the grammar is modified. This loss of fitness is especially bad at generation 20 because this is the first time the grammar is modified and when the grammar will undergo the largest changes to the original grammar. These large changes come from the addition of module library non-terminals to the original grammar. The degradations in fitness at each subsequent grammar modification step is much less because it is unlikely that a module library non-terminal will be added or removed from the grammar after they are introduced. The primary modifications being made are localized within the module library non-terminals. Using this particular approach, valuable information is being lost when the grammar is modified and GE appears unable to fully recover from the disruption caused by the remapping. On the other hand, in Figures 2(a) and 2(b) the *Repair* approach does not suffer from this drawback. When the grammar is changed, there is no loss in fitness at all, and evolution continues to progress.

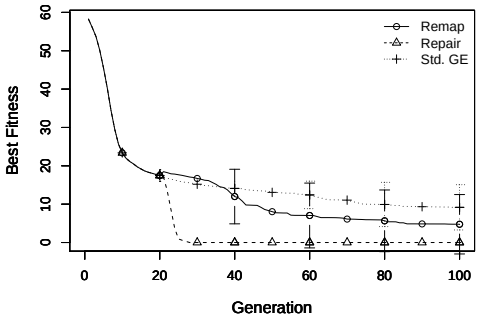
Note that the fitness of all approaches in Figures 2(c) – 2(e). Both the *Remap* and *Repair* approaches suffer **no** degradation in fitness when the grammar is changed, unlike the previous problems. They also



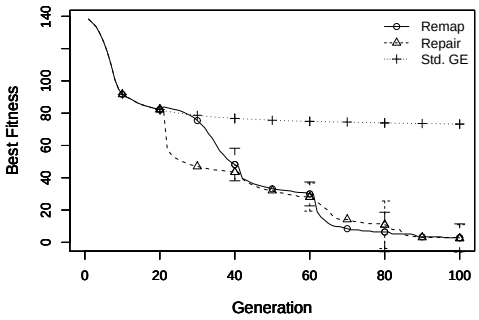
(a) Santa Fe Ant Trail Average Best Fitness



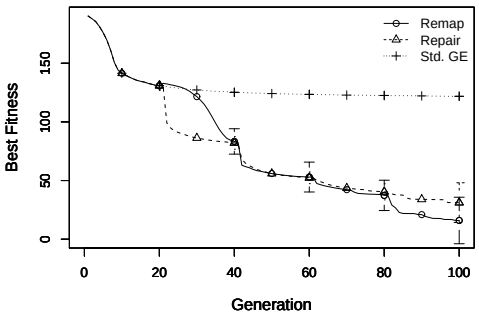
(b) Symbolic Regression Average Best Fitness (Note the log scale y-axis)



(c) 8×8 Lawn Mower Average Best Fitness



(d) 12×12 Lawn Mower Average Best Fitness



(e) 14×14 Lawn Mower Average Best Fitness

both significantly outperform standard GE. The reason for this is the amount of information compression that occurs when modules are added to the grammar and the size of the derivation trees required to reach a solution. Standard GE has to incrementally grow its individuals using crossover and mutation operations, making it more difficult to reach larger phenotypes needed to solve the problems. The grammar modification approaches encapsulate potentially large sub-derivation trees into a single production in the grammar. These large sub-derivation trees might require many codons to derive initially, but once they have been encapsulated into a module, they may be produced using only one or two codons.

A summary of these graphs is given in Table 2. This table shows the average best fitness for each approach at the end of their respective evolutionary runs. It further shows the merit of modifying the grammar and using the genotype repair approach. For every benchmark problem, the *Repair* approach boasts more successful runs than standard GE. It also finds more target solutions than the *Remap*

approach in three out of five benchmark problems, and for the remaining two problems the difference in successful runs is minimal at only two and one respectively. It also shows the *Remap* and *Repair* performing better than standard GE with statistically significant results on the Lawn Mower problems.

Table 2: This table summarizes the graphs in Figures 2(a) – 2(e) as well as gives the number of successful runs, paired t-tests for significance comparing the *Remap* and *Repair* approaches to standard GE, and a paired t-test comparing the *Remap* approach to the *Repair* approach. Note that a run for the Symbolic Regression benchmark is considered a success if the best fitness is less than or equal to 0.01. For every other benchmark, a run is a success if the best fitness is 0.

Approach	**Best Fitness** $\pm$ **Std. Dev.**	**P. Value (Std. GE)**	**P. Value (Remap)**	**Number Solved (Out of** 50**)**
Santa Fe Ant Trail				
Remap	19.10 ± 11.67	1	NA	8
Repair	13.98 ± 11.06	0.44	1	16
Std. GE	14.98 ± 10.04	NA	NA	12
Symbolic Regression ($x^5 - 2x^3 + x$)				
Remap	0.57 ± 0.81	1	NA	29
Repair	0.17 ± 0.94	0.19	1	41
Std. GE	0.29 ± 0.73	NA	NA	36
Lawn Mower (8×8)				
Remap	4.76 ± 7.74	0	NA	25
Repair	0 ± 0.001	0	1	49
Std. GE	9.17 ± 5.89	NA	NA	9
Lawn Mower (12×12)				
Remap	2.83 ± 8.61	0	NA	14
Repair	2.47 ± 8.49	0	1	12
Std. GE	73.24 ± 2.49	NA	NA	0
Lawn Mower (14×14)				
Remap	15.94 ± 19.83	0	NA	2
Repair	31.17 ± 16.91	0	0	1
Std. GE	121.81 ± 2.04	NA	NA	0

6.2 Grammar Modification and Information Compression

One advantage of being able to exploit modularity is the capability to compress beneficial information into a module and make that information more accessible for reuse during evolution. This can also be thought of as changing the bias of the grammar. The grammar modification approaches used in this work are notably good at compressing information for further reuse. For the sake of space, only the problem which best exhibits this is shown (the 12×12 Lawn Mower problem).

Figure 2 shows the average derivation tree depth per individual in the population. The graph of the average codons used in the population is similar, but has been omitted to save space. Figure 2 shows the size of the individuals' derivation trees races to the depth of 80 early in the evolutionary process. This is because phenotypes with relatively large numbers of terminal symbols are needed to hold enough information to solve this particular problem. The average used codons also increase with the derivation tree depth as they are tightly linked to one-another.

At generation 20 the *Remap* approach plummets in terms of the the size of derivation trees, but referring back to Figure 2(d) it is apparent that the average best fitness does not suffer. At this generation, the grammar is modified and modules are added, facilitating a more codon-efficient manner to express larger amounts of information. Simply put, this allows for more lawn-mowing instructions to be represented with fewer codons and more shallow derivation trees. The *Repair* approach also drops in codon usage and derivation tree size, but not nearly to the extent of the *Remap* approach. After the first module identification/grammar modification step, the size of derivation trees begin to increase and a wave-like pattern can be seen. This figure (Figure 2) suggests that when new modules are encapsulated,

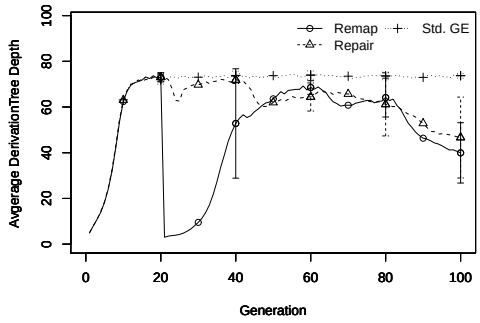


Figure 2: Average Derivation Tree Depth - 12 × 12 Lawn Mower Problem

they are likely to encompass an old module, or possibly multiple old modules, containing some amount of good phenotypic information. This information is then placed in the grammar and can be easily accessed by future individuals. Standard GE, on the other hand quickly reaches the limit for derivation tree depths and becomes stuck, both in the size of individuals and in fitness (again see Figure 2(d)). These results show how evolution is able to compress and reuse the good information encapsulated into modules and made more accessible through adding it to the grammar.

6.2.1 Module Example

The claim is made throughout this paper that modules are being used by individuals to find better solutions and compress good information during evolution. To back this claim up, an example of a frequently used module that appears in highly fit individuals is given. The module in Figure 3(a) appears in the 84^{th} generation of one run of the 12×12 Lawn Mower problem. In that generation, this module was used by 81.2% of the population and the average fitness of individuals containing this module was 0.1789. This module uses 271 terminal symbols and 542 codons to produce. After considering Figure 3(b), it is easy to imagine how large a genotype is needed to encode this amount of information as well as information that was not part of the module. By encapsulating all these terminal symbols into one module, such a large quantity of information may now be expressed using one or two codons (depending on how many modules are in a particular module library rule).

6.3 Grammar Modification Comparison

Now that the differences in performance have been described, more analysis will be given in order to understand the differences between the *Remap* and *Repair* approaches. The manner in which they impact how often modules are used in the population and the fitness of individuals they appear in is explained. To complete this analysis, Table 3 will be used. In this table, the lifetime of a module refers to how many generations a module appears, the proportion is the number of individuals which use a particular module, and the fitness is the fitness of individuals containing a particular module.

The first thing to notice in Table 3 is that for every problem, at least one of the modules that was encapsulated and used in the grammar was present in the optimal solution in at least one run. This suggests that even the best solutions are using at least one module. When this and Table 2 are considered together, it is reasonable to believe that both the *Remap* and *Repair* approaches are able to use modules to find better solutions.

The next aspect of this table to acknowledge is the proportion of the population that modules appear in. For each module-encapsulating approach, there are modules that get used in large proportions (92%–100%) of the population. This is easily explained in the *Remap* approach because when the grammar is modified, it is very likely that a large percentage of the individuals in the population will contain modules after they have been added to the grammar and all individuals have been remapped. This forces modules into the population at the expense of information developed over the generations leading up to the grammar modification. This is reflected in the best fitnesses achieved by this approach in

```
mow left mow mow mow mow right mow mow mow mow mow mow
right mow mow mow mow mow mow mow right mow mow mow
mow mow right mow mow mow mow mow mow mow mow mow mow
mow mow mow left mow mow mow mow mow mow mow mow mow
mow mow right mow mow mow mow mow mow mow right mow
mow mow mow mow right mow mow mow mow mow mow mow mow
mow mow mow right mow mow mow mow mow right mow mow
mow mow mow left mow mow mow right mow mow mow mow mow
mow mow mow mow mow mow left mow mow mow mow mow mow
right mow mow mow mow mow mow right mow mow mow mow mow
mow mow mow right mow mow mow mow mow mow mow mow mow
mow left right right mow mow mow mow mow mow left
mow mow mow mow mow mow left mow mow mow left mow
mow mow mow right mow right left right mow mow mow mow
right mow mow mow mow mow left mow mow mow right mow
mow mow mow mow mow mow left mow mow mow left mow mow
right left mow mow mow mow right mow mow mow mow mow
mow mow mow right mow mow mow mow right mow mow mow mow
mow mow mow mow mow mow left right right mow mow mow
mow mow mow mow left mow mow mow right mow mow right
mow mow mow mow mow mow mow mow right mow mow mow mow
mow mow mow mow
```

(a) The phenotype of an example module found for the 12×12 Lawn Mower problem

```
<prog> ::= <command> | <command> <prog>
<command> ::= mow | left | right
```

(b) The initial grammar used for all Lawn Mower problems

Figure 3

Table 2. With the *Repair* approach, modules are added to the grammar, but they are introduced into individuals through crossover and mutation operators, which is not destructive to the mapping and, depending on the location in the individual of the crossover and/or mutation, is not so destructive to the information previously assembled by evolution. This same reasoning also explains the differences between the two grammar modification approaches in the average proportion of individuals in which modules are found.

The final notable characteristic of this table is the lifetime of modules in the population. For the Santa Fe Ant Trail, Symbolic Regression, and 8×8 Lawn Mower problems, some modules are introduced at the first module identification/ grammar modification step and persist through the entire evolutionary run. This suggests that some of the early modules are beneficial enough to not be replaced or evolved out of the population. On the harder Lawn Mower problems, however, no module was ever discovered in 50 trials per experimental setup that lasted the longest possible number of generations. Given the nature of these problems (being able to express more phenotypic information with less genotypic information is beneficial), it is very probable that modules are being encapsulated into larger, better modules which eventually replace the initially discovered modules. This is enforced by the average lifetime of modules which is less on the 12×12 and 14×14 Lawn Mower problems than all the other problems.

7 Conclusion and Future Work

This paper presents two approaches to enhancing GE's grammar over the course of an evolutionary run (*Remap* and *Repair*) and compares them to standard GE. These new approaches identify potentially beneficial modules and add them to GE's grammar. The difference between them is that the *Remap* approach simply adds modules to the grammar and remaps the entire population using the new grammar, while the *Repair* approach ensures that all individual maintain their pre-grammar modification mapping. The results show that there are varying levels of merit in being able to encapsulate beneficial information and make it easily accessible through a grammar, depending on the problem. By simply changing the grammar and letting evolution take over, beneficial information can be destroyed, but by changing the grammar and ensuring that no information is disrupted during the grammar modification process, there are potential gains in performance.

The results of the research presented in this paper bring to mind a number of possible venues for future work. The first of these is allowing the identified modules to be modified and/or evolved by evolutionary

Table 3: This table shows the average lifetime of modules, average proportion of individuals a module is used in, and average fitness of individuals a module is present in (all ± Std. Dev.). It also shows the maximum lifetime a module existed in the population, the maximum proportion of individuals a module appeared in, and the average best fitness of an individual with at least one module in it.

	Lifetime		**Proportion**		**Fitness**	
Approach	**Avg.**	**Max.**	**Avg.**	**Max.**	**Avg.**	**Best**
Santa Fe Ant Trail						
Remap	41.12 ± 25.07	80.00	0.07 ± 0.14	0.99	58.58 ± 27.35	0.00
Repair	32.01 ± 23.52	79.00	0.03 ± 0.10	0.98	70.00 ± 17.02	0.00
Sym. Reg. $(x^5 - 2x^3 + x)$						
Remap	40.76 ± 22.56	80.00	0.08 ± 0.20	1.00	$2.12 \times 10^7 \pm 1.40 \times 10^9$	0.00
Repair	59.99 ± 21.41	79.00	0.04 ± 0.13	0.97	$5.39 \times 10^{13} \pm 2.87 \times 10^{15}$	0.00
Lawn Mower (8×8)						
Remap	35.81 ± 19.96	80.00	0.05 ± 0.17	0.95	15.53 ± 9.69	0.00
Repair	35.15 ± 16.19	79.00	0.04 ± 0.10	0.92	9.33 ± 10.15	0.00
Lawn Mower (12×12)						
Remap	17.68 ± 5.64	55.00	0.06 ± 0.18	0.94	48.22 ± 30.47	0.00
Repair	20.92 ± 9.67	65.00	0.04 ± 0.14	0.94	46.26 ± 24.43	0.00
Lawn Mower (14×14)						
Remap	16.51 ± 5.53	59.00	0.06 ± 0.19	0.96	84.22 ± 37.08	0.00
Repair	19.38 ± 9.11	59.00	0.05 ± 0.15	0.93	81.82 ± 29.57	0.00

operators like crossover and mutation. Identified modules may also be parameterized, creating a type of grammatical function. Other possibilities for future work include examining alternative methods for identifying modules and fine tuning the parameters used in Section 3. While identifying modules is not the primary focus of this paper, the ability to identify better modules could lead to improvements in the grammar modification methods used here. This paper tackled three different classes of problems, and to further test the limits of the grammar modification approaches more benchmark problems can tested, as well as dynamic and real world problems.

8 Acknowledgements

The authors would like to thank members of the UCD Natural Computing Research and Applications group for their support, comments, and discussions. This research is based upon works supported by the Science Foundation Ireland under Grant No. 08/RFP/CMS1115 and No. 08/IN.1/I1868.

References

[1] Peter J. Angeline and Jordan Pollack. Evolutionary module acquisition. In D. Fogel and W. Atmar, editors, *Proceedings of the Second Annual Conference on Evolutionary Programming*, pages 154–163, La Jolla, CA, USA, 25-26 February 1993.

[2] Peter J. Angeline and Jordan B. Pollack. The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pages 236–241, Bloomington, Indiana, USA, 1992. Lawrence Erlbaum.

[3] Ian Dempsey, Michael O'Neill, and Anthony Brabazon. *Foundations in Grammatical Evolution for Dynamic Environments*. Springer, 2009.

[4] Ozlem Garibay, Ivan Garibay, and Annie Wu. The modular genetic algorithm: Exploiting regularities in the problem space. *Computer and Information Sciences - ISCIS 2003*, pages 584–591, 2003.

[5] Robin Harper and Alan Blair. Dynamically defined functions in grammatical evolution. In *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 9188–9195, Vancouver, 6-21 July 2006. IEEE Press.

[6] Erik Hemberg. *An Exploration of Grammars in Grammatical Evolution*. PhD thesis, University College Dublin, 2010.

[7] Erik Hemberg, Conor Gilligan, Michael O'Neill, and Anthony Brabazon. A grammatical genetic programming approach to modularity in genetic algorithms. In Marc Ebner et al., editors, *EuroGP 2007: Proceedings of the 10th European Conference on Genetic Programming*, number 4445 in LNCS, Valencia, Spain, 2007. Springer.

[8] Erik Hemberg, Michael O'Neill, and Anthony Brabazon. An investigation into automatically defined function representations in grammatical evolution. In R. Matousek and L. Nolle, editors, *15th International Conference on Soft Computing, Mendel'09*, Brno, Czech Republic, 24-26 June 2009.

[9] John H. Holland. *Adaptation in natural and artificial systems*. The University of Michigan Press, Ann Arbor, 1975.

[10] Maarten Keijzer, Conor Ryan, and Mike Cattolico. Run transferable libraries —learning functional bias in problem domains. *Genetic and Evolutionary Computation –GECCO 2004*, pages 531–542, 2004.

[11] John R. Koza. *Genetic Programming: on the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[12] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, USA, 1994.

[13] Krzysztof Krawiec and Bartosz Wieloch. Functional modularity for genetic programming. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 995–1002, New York, NY, USA, 2009. ACM.

[14] Hammad Majeed and Conor Ryan. Context-aware mutation: a modular, context aware mutation operator for genetic programming. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1651–1658, New York, NY, USA, 2007. ACM.

[15] M. O'Neill, E. Hemberg, C. Gilligan, E. Bartley, J. McDermott, and A. Brabazon. GEVA: grammatical evolution in Java. *ACM SIGEVOlution*, 3(2):17–22, 2008.

[16] Michael O'Neill and Conor Ryan. Grammar based function definition in grammatical evolution. In *GECCO '00: Proceedings of the Genetic and evolutionary computation Conference*, pages 485–490, 2000.

[17] Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers, 2003.

[18] Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, and Wolfgang Banzhaf. Open issues in genetic programming. *Genetic Programming and Evolvable Machines*, 11:339–363, 2010. 10.1007/s10710-010-9113-2.

[19] Riccardo Poli. Parallel distributed genetic programming. Technical Report CSRP-96-15, School of Computer Science, University of Birmingham, B15 2TT, UK, September 1996.

[20] Conor Ryan, Maarten Keijzer, and Mike Cattolico. Favourable biasing of function sets using run transferable libraries. *Genetic Programming Theory and Practice II*, pages 103–120, 2005.

[21] Herbert A. Simon. *The sciences of the artificial*. MIT Press, 3 edition, 1996.

[22] John Mark Swafford, Michael O'Neill, Miguel Nicolau, and Anthony Brabazon. Exploring grammatical modification with modules in grammatical evolution. In Sara Silva, James A. Foster, Miguel Nicolau, Penousal Machado, and Mario Giacobini, editors, *EuroGP*, volume 6621 of *Lecture Notes in Computer Science*, pages 310–321. Springer, 2011.

[23] J.A. Walker and J.F. Miller. The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. *Evolutionary Computation, IEEE Transactions on*, 12(4):397 –417, August 2008.

[24] P.A. Whigham. Inductive bias and genetic programming. In *Genetic Algorithms in Engineering Systems: Innovations and Applications, 1995. GALESIA. First International Conference on (Conf. Publ. No. 414)*, pages 461 –466, September 1995.