Exploring Grammatical Modification with Modules in Grammatical Evolution

John Mark Swafford, 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

There have been many approaches to modularity in the field of evolutionary computation, each tailored to function with a particular representation. This research examines one approach to modularity and grammar modification with a grammar-based approach to genetic programming, grammatical evolution (GE). Here, GE's grammar was modified over the course of an evolutionary run with modules in order to facilitate their appearance in the population. This is the first step in what will be a series of analysis on methods of modifying GE's grammar to enhance evolutionary performance. The results show that identifying modules and using them to modify GE's grammar can have a negative effect on search performance when done improperly. But, if undertaken thoughtfully, there are possible benefits to dynamically enhancing the grammar with modules identified during evolution.

1 Introduction

Modularity in evolutionary computation has been studied in a variety of contexts. The range of research covering this topic starts with principles taken from biology [17] and extends to the empirical analysis of performance of different approaches to enabling and exploiting modularity. The work presented here 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.

As genetic algorithms (GAs) [8] and genetic programming (GP) [10] have been studied fairly extensively in this context, this work focuses on grammatical evolution (GE) [14]. Studying modularity in the context of GE is especially interesting because of the genotype-to-phenotype mapping process employed. The context-free grammar used in this process provides one of the easiest methods of reusing information in GE, incorporating that information into the grammar. By examining the derivation trees created by this mapping process, "good" information found by GE can be encapsulated and placed directly in the grammar to be used in the correct context. However, modifying the grammar in this way comes with

possible drawbacks, which will be discussed later, in Sect. 3. As modularity in GP is one of the key open issues [15], the research presented here aims to add to the knowledge base of enabling modularity in grammar-based GP. This is accomplished by exploring initial methods of identifying, encapsulating, and reusing modules in GE by modifying the grammar during an evolutionary run.

In this paper, a module refers to a sub-derivation tree of an individual which is considered to contain beneficial information. It is also important to point out that there are a few differences between this approach and using Koza's automatically defined functions (ADFs) [12]. Individuals using ADFs each have their own, local, ADFs, which are may be modified by genetic operators. Modules in this study are stored in the universal grammar for all individuals. Once encapsulated in the grammar, they may not be modified by genetic operations.

The rest of the paper is structured as follows. Next, Section 2 outlines some previous work relating to modularity in GP and GE. Section 3 presents the experimental setup used here, and Sect. 4 details the results of this work, and their meaning. Finally, Sect. 5 gives the conclusions and avenues for future work.

2 Previous Work

Some of the earliest and most relevant work is Angeline and Pollack's [1, 2] 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* to encapsulate sub-trees into modules, release entire or pieces of compressed sub-trees, and make compressed modules part of the original representation. Angeline and Pollack 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.

Another relevant body of work is that conducted by Keijzer et al. [9]. They introduce the notion of run-transferable libraries (RTLs), which are lists of modules discovered over a number of independent runs and used to seed the population of a separate 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 [16].

In the context of GE, modularity has been previously studied in a variety of ways. Hemberg et al. [6] use 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),[3] on problems known to have regularities (one example is the checkerboard problem).

The most popular approach to enabling and exploiting modularity in GP is the use of automatically defined functions (ADFs) [12]. ADFs are parameterized functions which are evolvable and reusable sub-trees in a GP individual. GP equipped with this form of modularity is shown to out-perform standard GP on problems of sufficient difficulty [12]. There have been previous approaches to enabling ADFs, in GP [11] and GE [7] (as well as the similar Dynamically Defined Functions in GE [4]). A more in-depth review of previous work modularity in GP can be found in work by Walker and Miller [18] and Hemberg [5].

3 Experimental Setup

The purpose of this work is to examine a simple method of identifying modules in GE and using these modules to enhance GE's grammar, and subsequently performance. The first step in this process is picking the parent individual from which a module will be selected. Once the parents are selected, candidate modules are picked from these parents and evaluated. After the modules have been evaluated, they may (or may not) go through a replacement operation which throws away modules based on a fitness value assigned when each module is evaluated. Finally, the collected modules are added to the grammar and the main evolutionary loop continues. Figure 1 shows an example of this process.

Module Identification: There are a number of steps involved in picking out suitable modules. The first step in identifying a module is to pick a parent individual which will provide a candidate module. All experiments presented in this work use an iterative approach where each candidate module

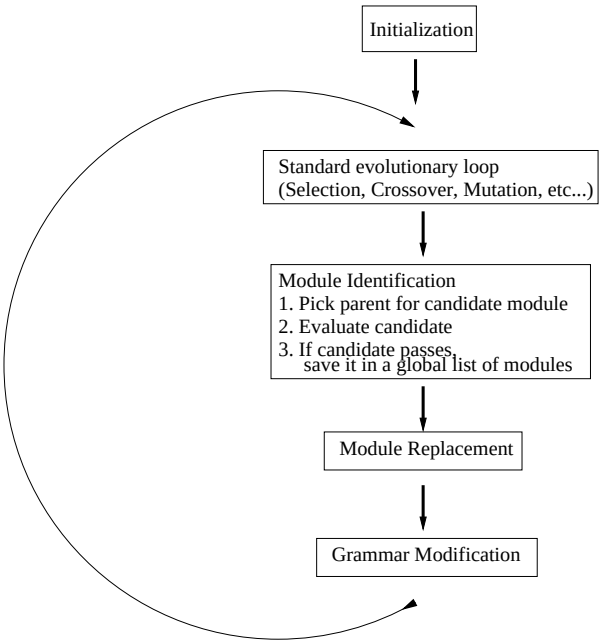


Figure 1: Evolutionary loop with module identification and grammar modification

was taken from each individual in the population. A roulette selection for the parents was tested in preliminary experiments, but the results of these tests showed no difference between the roulette and iterative approaches.

The next step in the module identification process is picking a candidate module and evaluating it. To do this, a random node on the parent individual's derivation tree is picked. The sub-derivation tree starting at this node is the candidate module, and original fitness of the individual, f_0, is recorded. Next, 50 new derivation tree 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, $f_{1...50}$. If f_0 is a better fitness than 75% of $f_{1...50}$, the candidate module is saved for later use. When this is the case, the difference between f_0 and each of $f_{1...50}$ is taken and the average of these is the module's fitness value. No attempt was made to optimize these parameters. This approach to module identification was inspired by that taken by Majeed and Ryan [13].

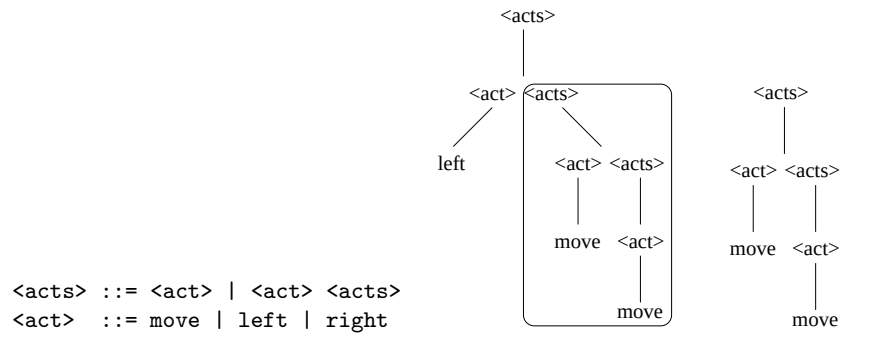
If left unchecked, the list of modules could grow to unreasonable sizes. In order to remedy this, only the 20 best modules are kept after each module identification step, before using them to modify the grammar. The problem used in this work was the Santa Fe ant trail as it has been shown to benefit from other approaches to modularity, both in standard GP [12] and GE [7]. The experimental setup can be found in Table 1. The context-sensitive crossover and mutation in Table 1 are similar to GP's sub-tree mutation and crossover, except they operate on GE derivation trees. These operators ensure that when sub-trees are crossed the appropriate portions of the genotype are also swapped.

Grammar Enhancement by Modules: Once modules have been identified, they must somehow be incorporated back into the evolving population. This was done by simply adding the modules to the appropriate rules in the grammar. This was done two different ways. First, consider the simple grammar in Fig. 2(a), an individual producible by that grammar (Fig. 2(b)), and a module selected from that individual (Fig. 2(c)). This module is incorporated into the grammar by taking the module's phenotype (`move move`) and making it a production of the rule matching the module's root symbol. Since the module's root symbol is `<acts>`, the module will be added to the rule

```
<acts> ::= <act> | <act> <acts>.
```

The updated rule would then be

```
<acts> ::= <act> | <act> <acts> | <mod_0>,
```



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

Figure 2: A sample grammar, an individual producible by that grammar, and a sample module from that individual. Note: the grammar in Fig. 2(a) is not the actual grammar used in this work.

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

(a) Grammar not using module libraries

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

(b) Grammar using module libraries

Figure 3: Grammars not using and using module libraries

and a new rule would be created:

```
<mod_0> ::= move move.
```

The complete grammar with this modification can be seen in Fig. 3(a). Given the nature of the phenotype to genotype mapping in GE, it is quite obvious that modifying the grammar in this manner has the potential to be extremely destructive. When a new production is added to any rule in the grammar, is it likely that individuals using that rule will no longer map to the same phenotype. An alternative, less destructive, method was also used to modify the grammars. In this approach, module library non-terminals were added to the grammar instead of adding the module directly. Again, consider the initial grammar (Fig. 2(a)), individual (Fig. 2(b)), and module (Fig. 2(c)). Using the module library method, a new non-terminal will be added to the rule matching the module's root node:

```
<acts> ::= <act> | <act> <acts> | <acts_mod_lib>,
```

and a new rule is created for the actual module phenotypes:

```
<acts_mod_lib> ::= <mod_0>,
```

and a new rule will be created:

```
<mod_0> ::= move move.
```

The grammar modified in this way can be seen in Fig. 3(b). By using the module library non-terminals, the addition and subtraction of modules is localized to those library non-terminals. This should have the effect of less disruption to the individuals when the grammar is being modified and they are remapped.

Modifying GE's grammar based on sub-trees which are considered valuable is similar to Whigham's [19] work, where he also identifies what are considered to be useful sections of parse trees and alters the grammar with additional productions containing the terminals and possibly non-terminals from these parse trees. However, his grammar-based form of GP does not employ the genotype-to-phenotype mapping used in GE.

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

Parameter	Value
Generations	100
Population	500
Selection	Tournament (Size 5)
Wrapping	None
Crossover	Context-sensitive (80%)
Mutation	Context-sensitive (20%)
Elites	5
Initialization	Ramped Half and Half
Replacement	Generational
Max. Derivation Tree Size	50

4 Results and Discussion

This section explains how the different approaches to module identification and grammar modification impact GE's search ability. The effects of grammar modification using modules are shown and analyzed in terms of their impact on the fitness of the population over an evolutionary run.

4.1 Grammar Modification

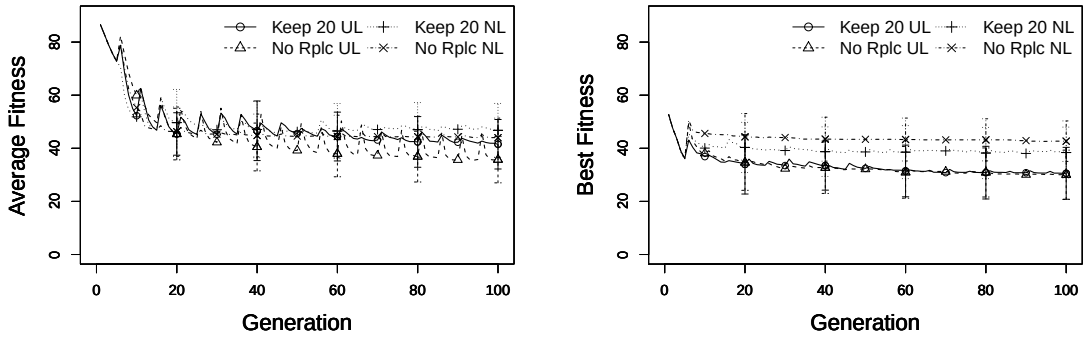
One of the factors that can make a large impact on how destructive or beneficial grammar modification can be is the manner in which the grammar is modified. In this work, there are two techniques for doing this: 1) adding modules as productions to their respective rules in the grammar, and 2) using module library rules to incorporate modules into the grammar (See Sect. 3).

The graphs in Figs. 4 and 5 show the differences between four approaches to adding modules to the grammar. First, only the best 20 modules were added to the grammar using module libraries. Next, all modules were added using module libraries. The same setups were repeated, but without the use of module libraries. The purpose for showing these particular variations is to see how adding small and large numbers of modules to the grammar in different ways affect the population's fitness. Figures 4(a) and 4(b) suggest that when the grammar is modified frequently (every 5 generations), approaches using module libraries to modify the grammar are more resistant to the dis-improvement in fitness (both the population's average and best fitness) that can come with remapping the population, but when the grammar is modified less frequently (Figs. 4(c) – 4(d) show grammar modification every 20 generations), approaches that do not use module libraries perform better.

Another noteworthy feature of Fig. 4 is the consistent way in which the fitness of the population is affected between the different setups when the grammar is modified at different intervals. The best fitness of runs where the grammar is modified every 5 generations is not as good as the runs where the grammar is only modified every 10 generations. The same applies for runs where the grammar is modified every 10 and 20 generations (The figures showing the fitness of modifying the grammar every 10 generations was omitted for lack of space). This can be thought of in terms of evolution not having enough time to exploit the new grammar when grammar modification intervals are short.

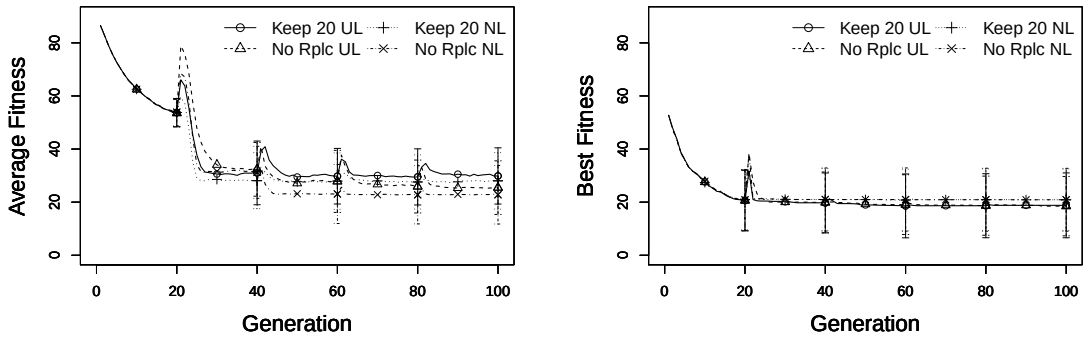
When modules are found and added to the grammar, individuals must be re-mapped in order to use them. With GE's mapping process the addition of just one production to any rule in the grammar has the possibility to cause an individual to map to a completely different phenotype, thus losing any valuable information that individual might have developed so far. One way to think of this is the population going into a state of "shock" when the grammar is modified. This shock has a negative impact on the fitness of all the individuals, and the population needs time to recover in order for the best fitness to continually improve. As can be seen in the graphs in Figs. 4(a) and 4(c) the longer the gap between shocks the longer the population has time to recover and improve its fitness, but when it is shocked more regularly, this recovery period is shortened and the population is unable to improve to its maximum potential.

To further examine these setups, Fig. 5 shows distributions of how the average fitness of the population changes when the grammar is modified. Examining this figure shows that changing the grammar is



(a) Average Fitness

(b) Best Fitness



(c) Average Fitness

(d) Best Fitness

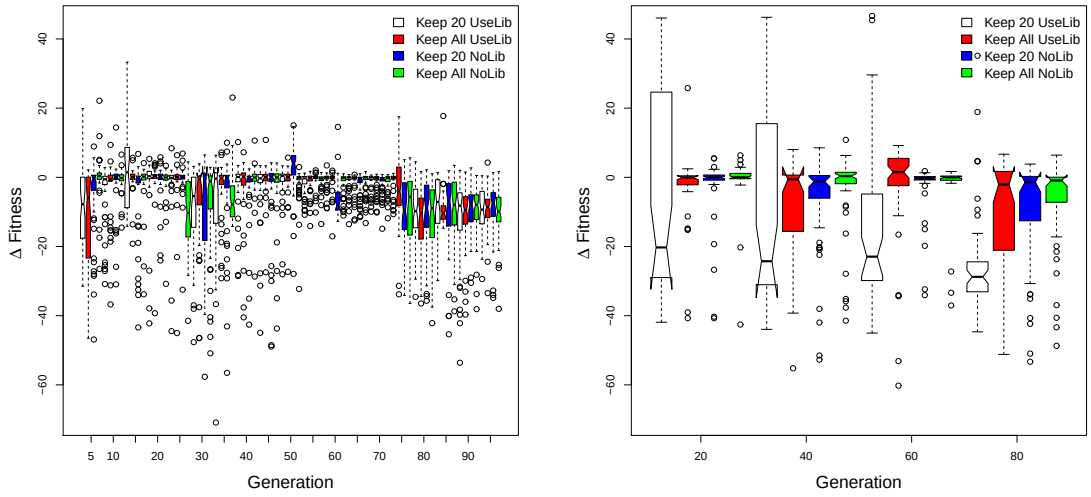
Figure 4: Best and average population fitness values for the Santa-Fe ant trail problem. Figures 4(a) and 4(b) show the grammar changing every 5 generations. Figures 4(c) and 4(d) show the grammar changing every 20 generations.

very likely to have a negative impact on the fitness of the population, especially when the grammar is changed frequently. As there is no consistent significant difference in the approaches here in terms of the degradation of fitness when the grammar is modified, this raises the question if there is some way to modify the grammar in order to minimize this loss of good information. To help explain the above results, one aspect of the different approaches to consider is how many modules are found per identification step as this affects the number of productions being added to the grammar. Note that this is not how many modules are actually used to enhance the grammar, just how many are identified before the module replacement takes place. Figures 6(a) shows the average number of modules found at each module identification step. Figures for module identification steps 10 and 20 were omitted as they showed a similar trend, with the most modules being discovered at the first identification step and less at the subsequent identification steps. The violin plots in Fig. 6(b) show the distribution of these values.

Results so far show the largest differences between grammar modification with and without module libraries occur when the grammar is modified frequently. Since modifying the grammar with module libraries was the better of the two approaches (it is only significantly better when the grammar is modified frequently), it will be discussed in future sections, unless otherwise noted.

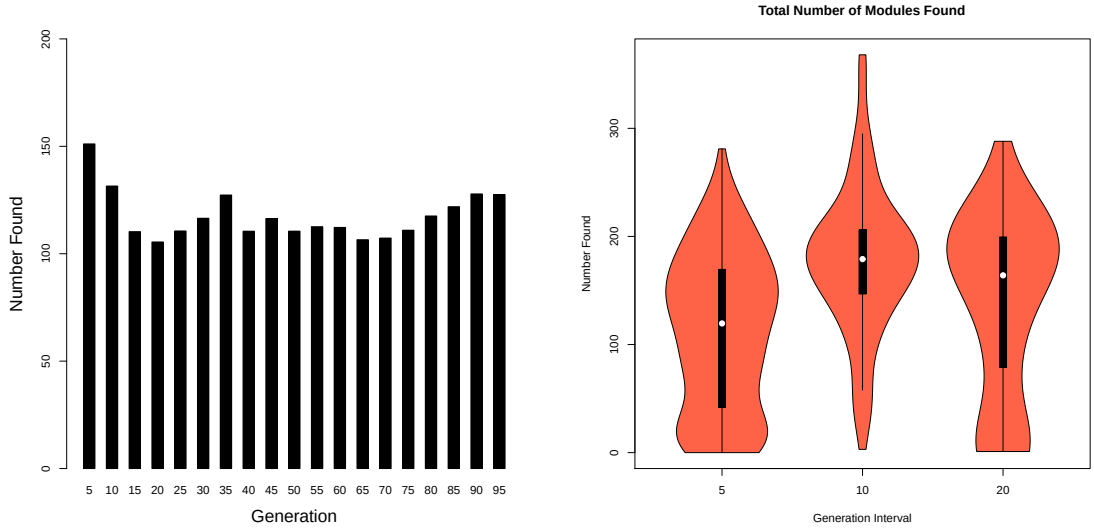
4.2 Grammar Enhancement and Fitness

Now, with some understanding of how modifying the grammar can change GE's search potential, the module-grammar enhancement technique is compared to standard GE. Fig. 7 shows the best and average



(a) (b)

Figure 5: Change in the population fitness when the grammar is modified. A negative change shows the fitness getting worse, while a positive change shows the fitness improving. The graph showing the change in average fitness every 10 generations was omitted for lack of space.

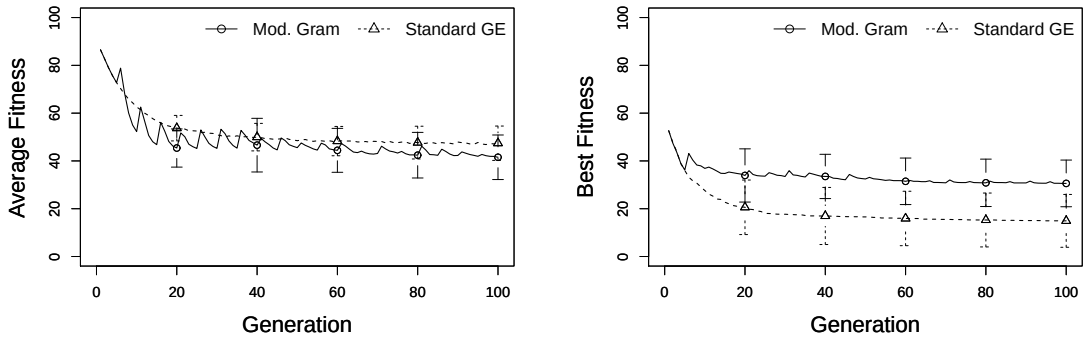


(a) Number of modules found per ID step

(b) Densities of the number of modules found with the different ID steps

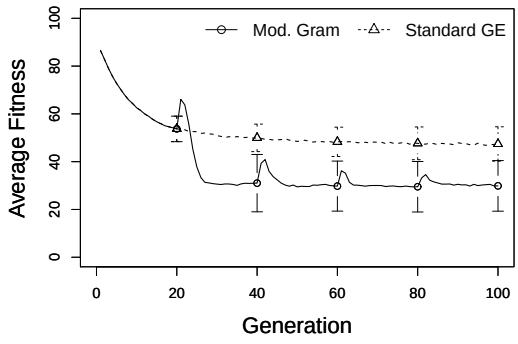
Figure 6: Figures 6(a) shows the average number of modules found per module identification step. Figure 6(b) shows the distribution of modules found with the different module identification intervals.

fitness of the grammar modification runs and the standard GE runs. This shows that modifying the grammar frequently is significantly detrimental to the performance of GE. But, if given longer intervals between grammar modification, the impact on average and best fitness is less notable. To further test the intervals between grammar modification and how they might impact search performance, longer runs were used with larger intervals between modifying the grammar. Fig. 8 shows runs with 500 generations and grammar modification steps of 20 and 100 generations. Figs. 8(a) and 8(b) show how maintaining the grammar modification step from earlier runs, but over a longer number of generations, does not offer much improvement in the best of average fitness of the individuals. But, when longer intervals between

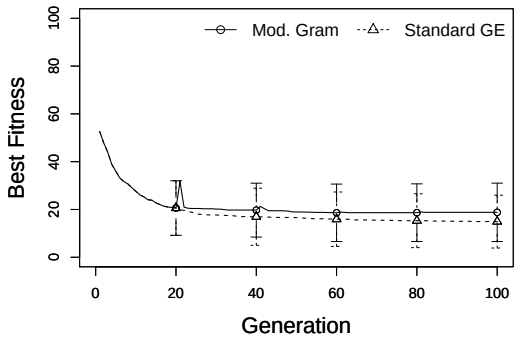


(a) Average Fitness

(b) Best Fitness



(c) Average Fitness



(d) Best Fitness

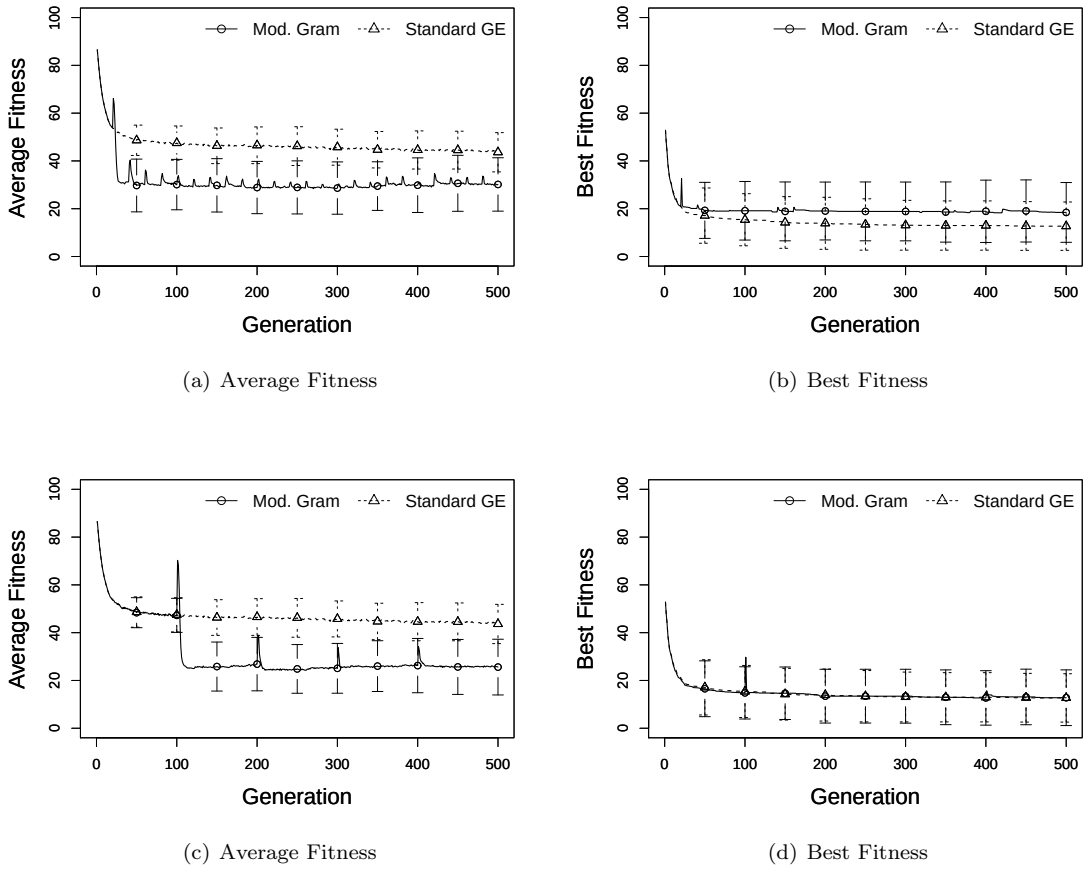
Figure 7: Comparisons of the average and best fitness values found by modifying the grammar every 20 and 5 generations and by standard GE. Figures 7(a) and 7(b) show the average and best fitness when the grammar is modified every 5 generations. Figures 7(c) and 7(d) show the average and best fitness when the grammar is modified every 20 generations.

grammar modification are used, a performance increase in the grammar modification approach can be seen. Figures. 8(c) and 8(d) show that the best performance, both for the average and best fitness of individuals, come when the grammar is modified at the longest interval of 100 generations. The difference in fitness values between approaches at the last generation suggests that modifying the grammar under these particular settings can have a beneficial effect on the population as a whole, without losing the valuable information in the best individual.

This data demonstrates that there are some benefits to adding modules to the grammar, but when the grammar is modified too frequently, or if the method of modifying the grammar does not make new and useful information easily accessible, there is no benefit in modifying the grammar, even with information that could be useful.

5 Conclusion and Future Work

This research examined one approach to identifying modules and using those modules to enhance the grammar in different ways with hopes of improving search in GE. There were some very distinct performance changes when the grammar was modified, which suggests that there is some merit to enhancing the grammar with information identified over the course of an evolutionary run. The results show that there is definitely a balancing act between modifying the grammar more or less frequently. It is also apparent that care must be taken when modifying the grammar in order to preserve the valuable infor-



(a) Average Fitness

(b) Best Fitness

(c) Average Fitness

(d) Best Fitness

Figure 8: Comparisons of the average and best fitness values found by modifying the grammar every 20 and 100 generations and by standard GE. Figures 8(a) and 8(b) show the average and best fitness when the grammar is modified every 100 generations. Figures 8(c) and 8(d) show the average and best fitness when the grammar is modified every 100 generations.

mation that already exists in the population. However, to truly understand the effects of modifying the grammar additional problems must be studied.

One of the major downfalls of this approach is that modifying the grammar creates a large negative impact on the fitness of the population and valuable information that has been learned will likely be lost. The first promising venue for future work includes identifying and implementing less destructive ways of modifying the grammar. Another possibility is making the modules found parameterized and/or capable of being modified by crossover and mutation operators. Even more possibilities for future work exist in the manner in which modules are identified. Here, only one approach was examined, but there are numerous other possibilities for module identification which may lead to discovering better modules and minimizing the loss of information when these modules are used to modify GE's grammar. This paper on presents the results for the Santa Fe ant trail problem. Future work will also include using more benchmark problems.

5.0.1 Acknowledgements:

The authors would like to thank members of the Natural Computing Research and Applications group for their support, comments, and discussions, especially Dr. Erik Hemberg. This research is based upon works supported by the Science Foundation Ireland under Grants No. 08/RFP/CMS1115 and 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] 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.

[4] 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.

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

[6] 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.

[7] 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.

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

[9] 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.

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

[11] John R. Koza. Architecture-altering operations for evolving the architecture of a multi-part program in genetic programming. Technical report, Stanford, CA, USA, 1994.

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

[13] 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.

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.